Università degli Studi della Calabria

Dipartimento di Ingegneria Informatica, Modellistica, Elettronica e Sistemistica

---

Computer Engineering for the Internet of Things

Project Report

Application of Big Data techniques and machine learning algorithms on environmental data using Weka and Apache Spark

for the exam

Big Data Analytics

Professor

Prof. Andrea Tagarelli
Ing. Lucio La Cava

Student

Alfredo Carnevale - 224433
Fabrizio Gabriele - 227506
Mario Pingitore - 225638

June 2023

Academic Year 2022-2023

Contents

1 Introduction

Big Data plays a critical role in the field of Internet of Things. Big Data and IoT come together in multiple ways. One of the most important concepts of IoT and Big Data combined is a **data generation**. IoT devices generate tons of data using sensors, actuators and other connected components. All generated data can include status information such as temperature, humidity, location and more. The massive volume of data requires proper administration and analysis.
The **data collection**, in this field, is therefore very important. Big data technologies make it possible to obtain and combine data from IoT sources. This means collecting informations and data from distributed gateways and devices, sometimes in real-time, and transferring it to a processing hub. Also the **data storage** plays an important role in the interaction between Big Data and IoT. This type of data is often kept in big data storage platforms like distributed file systems and data lakes. These solutions offer scalability, fault tolerance, and the ability to manage huge quantities of data.
Big data analytics techniques are deployed to extract beneficial information from IoT data. In order to find patterns, trends, correlations and anomalies, the data generated must be processed and analysed. Advanced analytics methods such as machine learning and predictive analysis can be used to make predictions and optimise IoT systems. Real-time analytics are frequently needed for IoT applications in order to enable quick actions and reactions. Big data platforms facilitate real-time processing and analysis of streaming data, enabling quick automation and decision making.

1.1 Big Data and Smart Building

Big Data IoT and Smart Buildings together provide a number of advantages in terms of productivity, sustainability and user experience.
Sensors and other devices that collect data on energy use, occupancy levels, ambient conditions, and equipment performance are installed in smart building. Big Data analytics makes it possible to manage and evaluate this data in order to find trends and insights. Building administrators can make decisions to optimise energy consumption, space utilization, and maintenance activities.
With Big Data IoT it's possible to track and analyze energy usage in Smart Buildings in real-time. Using a huge amount of data collected by sensors, tenants and administrator of the building may develop a picture and trends of energy usage patterns. The data-driven methodology, cited before, aids to identify ares of immoderate energy wastage, and to implement energy-saving measures. It may additionally assist with demand response scehmas, which include building modifying their energy use in response to grid conditions and pricing signals.
Smart buildings leverage Big Data IoT in order to carry out predictive maintenance strategies. Building tenants and operators can identify possible equipment failures before they happen by continuous monitoring building performance and gathering information on parameters like temperature, humidity and vibration. Predictive analytics algorithms can analyze these data to identify early warning signs, trigger maintenance alerts and schedule repairs or replacement proactively.
Big Data IoT allows a personalized and comfortable environment for building occupants. Smart Building can automatically modify settings to maximize tenants comfort by evaluating data on occupancy patterns, temperature preferences, lighting conditions.

1.2 Project purposes

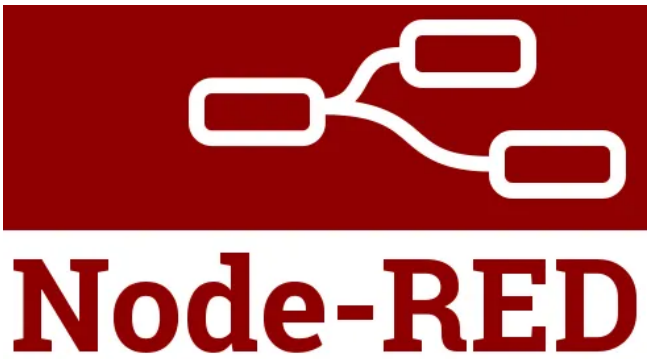
The first step of the project is to analyze the supplied dataset on **Weka**, to better understand how the tool works and to get informations about the classification and clustering algorithms that after we apply to our dataset.
Data have been sensed and collected from the environment, enriched with infos coming from a well-known weather webservice and used to produce a .csv dataset. This dataset has been then examined using Weka tool in order to find interesting patterns and eventual correlations. The project also implements a machine learning algorithm, run on an **Apache Spark** server, whose purpose is to enable fault-tolerance for the system. In fact, if some fail from some sensors happens, with consequential data loss, the system will exploit this algorithm, trained with the dataset composed by previous readings, to predict the value of the missed reading.

2 Hardware and Software

2.1 Weka

Weka is a well-liked open-source machine learning and data mining software tool that supplies a comprehensive set of algorithms and tools for data analysis. Weka has an intuitive graphical interface, making it usable by both beginners and experienced data scientist. It offers several data preparation tasks, visualization strategies and a variety of machine learning algorithms for classification, regression, clustering, association rules and feature selection.
One of the strength points of Weka is in its extensive collection of machine learning algorithms. It provides popular methods such as decision trees, random forests, k-nearest neighbors (KNN), Naive Bayes and neural networks.
Weka simplifies each step of machine learning from data pre-processing to model evaluation. It provides a lot of data transformation capabilities, including filtering, attribute selection, and instance sampling. Users can visualize data distributions, explore relationships between variables, and identify outliers or missing values.
Weka also offers tools for evaluating and comparing models using several values such as accuracy, precision, recall, and F1 score. It supports cross-validation, training and testing on separate datasets, as well as methods for tuning model parameters to optimize performance. The tool facilitates the integration of various tools and libraries in addition to its basic functions. This makes it possible for users to increase its functionality or incorporate it with other applications for certain needs.
Weka has gained popularity in both academia and industry due to its ease of use, flexibility and extensive documentation. It has been extensively used in a variety of fields, including as financing, healthcare, marketing and social sciences for educational purposes, research project and reals word application.
Overall, Weka is a flexible and powerful tool for machine learning and data mining, offering a comprehensive selection of features and algorithms that let users explore, analyze, and build predictive models from their data.

2.2 Node-RED



Node-RED is a powerful general-purpose development tool for visual programming applied to Web-of-Things applications. Node-RED exploits the *flow-based paradygm* **(FBP)** to connect together hardware devices, APIs and Web Services. Developed at the beginning of 2013 by IBM using a flexible JavaScript framework called Node.js and distributed for a wide variety of platforms, Node-RED is among the most valid tools available for web-based

IoT app developers since it allows to develop projects in a fast and concise way. The main concepts that must be absorbed,in order to understand how programming with Node-RED is like, are the *node* and the *flow*.
The **node** is an atomic element equipped with one or more input/output ports and containing some specific or custom code inside. In most cases nodes implement a *black box* model: is not possible to know the details of the code implemented inside, but only the input/output specifics. Node-RED provides a very rich and operational basic set of nodes, each one specialized in a single task and graphically diversified from the others with different colors and icons. It is also possible to add variety to the initial palette of nodes by installing new modules, in order to cover more protocols and add other functionalities.
Nodes can exchange messages each other through *wires* that start from the output port of a node and end in the input port of another node (it is noticeable how the Node-RED logo itself represents some nodes linked together by wires). Messages are simple JavaScript objects that can be tailored by the developer and are composed by two main properties:

- *payload*, containing the message itself;
- *topic*, that describes the subject of the message.

A **flow** is a collection of nodes that are linked together in a particular structure in order to accomplish a specific goal. Different flows can communicate each other thanks to particular nodes called **links** that create interconnections between them.
An important consideration, for the developer that approaches to Node-RED, has to be done to the variable management. **Variables** are mainly created inside **function** nodes and they can be initialized with three different *scope* levels:

- *context* is the default level, and it limits the visibility of the variable within the node in which it was created;
- *flow* is the intermediate level, allowing the variable to be seen and reused in any node belonging to the same flow in which the variable was created;
- *global* is the highest level and it allows the exchange of variables among different flows.

Variables and messages have a similar task, which is transmitting and sharing info among different nodes, but their functioning is very different, since messages are time-related event-based entities that can only travel from the node in which they were generated to all the other directly linked nodes (through wires), while variables can potentially share info among any sets of nodes but they can't be used to trigger other nodes.
Node-RED today can rely on a large community of users and is certainly one of the most successful programming tools for IoT projects, with its 2.0 version, released in 2021, that makes it more powerful and secure.

2.3 Apache Spark

Apache Spark is an open-source framework that is utilized usually for handling Big Data Application and to boost their performances. The big advantage of utilizing Spark is that the data that has to be analysed can be stored and processed in memory instead of utilizing disk-based alternatives and, moreover, they can also be processed in parallel. One of the key improvements with regard of its predecessor, Apache Hadoop MapReduce, is the ability of Spark of processing and storing the data in memory, while for example Hadoop implements a write/read operation from the disk: obviously, there has to be enough resource available to doing so and that's why Spark applications are often deployed on big cluster, to take full advantage of all the power available and to better achieve parallelism.
Apache Spark has a hierarchical master/slave architecture. The master node is the Spark Driver and is the one in charge of handling the Cluster Manager which, in turn, manages the worker (slave) nodes and delivers data results to the application client.

The Spark Driver generates the SparkContext, which works with the cluster manage (Spark's Standalone Cluster Manager or other cluster managers like Hadoop YARN, Kubernetes), to distribute and monitor execution across the nodes and also creates Resilient Distributed Datasets (RDDs), which are fundamental element that give to Spark its remarkable processing speed. Other data structures are the DataFrame (a row/column table) and the DataSet, an extension of the DataFrame.
Resilient Distributed Datasets (RDDs) are fault-tolerant collections of elements that can be distributed among multiple nodes in a cluster and worked on in parallel. Spark works by loading data (from a data source or an existing collection) in the RDD, that is located in memory and so all the transformations and processing operations on the data are performed in the memory: the data remain stored in memory until the system run out of memory or until the user decides to write the data to the disk for persistence.
The fundamental layer in the Spark architecture is composed by the **Spark Core**, which is the underlying general execution engine for spark platform that all other functionality is built upon. It provides In-Memory computing and referencing datasets in external storage systems and is the base for all parallel data processing and handles scheduling, optimization, RDD, and data abstraction; moreover provides, moreover, the functional foundation for the various **Spark APIs**.
Spark includes a variety of application programming interfaces (APIs) to help the development of a broader range of applications, such as Spark SQL, Spark Graphx, Spark Streaming and Spark MLlib. Following, a focus on the modules utilized in this project:

- The **Spark SQL** module is used to perform SQL-like operations on the data stored in memory. It is possible to either leverage using programming API to query the data or use the ANSI SQL queries similar to RDBMS. It is also possible to mix both, for example, and use API on the result of an SQL query. This way, a native support for SQL in Spark is achieved and the querying operations can be performed on both the RDDs and/or relational tables, other than providing all the perks of the Spark engine that allow to scale to thousand of nodes, perform multi-hour queries and having a full mid-query fault tolerance;
- **Spark MLlib** is the main module utilized in the Spark application in exam. MLlib is a scalable machine learning library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, and underlying optimization primitives. Spark MLLib seamlessly integrates with other Spark components such as Spark SQL, Spark Streaming. The module allows for preprocessing, munging, training of models, and making predictions at scale on data and is eve possible to use models trained in MLlib to make predictions in Structured Streaming. The library can interoperate with Numpy, R, and it provides tools such as:
  - *ML Algorithms*: common learning algorithms such as classification, regression, clustering, and collaborative filtering;
  - *Featurization*: feature extraction, transformation, dimensionality reduction, and selection;
  - *Pipelines*: tools for constructing, evaluating, and tuning ML Pipelines;
  - *Persistence*: saving and load algorithms, models, and Pipelines;
  - *Utilities*: linear algebra, statistics, data handling, etc.

2.4 Apache Kafka

Apache Kafka is an event streaming platform used to collect, process, store, and integrate data at scale. It has numerous use cases including distributed logging, stream processing, data integration, and pub/sub messaging. Kafka is based on the abstraction of a distributed commit log. By splitting a log into partitions, Kafka can scale-out systems. As such, Kafka models events as key/value pairs: internally, keys and values are just sequences of bytes,

but externally they are often structured objects represented the programming language used. Kafka calls the translation between language types and internal bytes serialization and deserialization. The serialized format is usually JSON, JSON Schema, Avro, or Protobuf. Apache Kafka is based on some key elements:

- **Topic**. A topic can be seen as an immutable log of events and can be configured to expire data after it has reached a certain age (or the topic overall has reached a certain size), from as short as seconds to as long as years or even to retain messages indefinitely. The logs that underlie Kafka topics are files stored on disk, so when an event is written on a topic, it is as durable as it would be written to any persisting trusted database. This way, the work of storing messages, writing new messages, and processing existing messages can be split among many nodes. The single topic can also be partitioned into multiple logs, each of which can live on a separate node in the Kafka cluster.
- **Broker** The Brokers are independent machines each running the Kafka broker process. Each broker hosts some set of partitions and handles incoming requests to write new events to those partitions or read events from them. Brokers also handle replication of partitions between each other. A topic (or a partition) can also be replicated on multiple brokers: the original is called leader replica, while the copies are called follower replicas.
- **Producer** In a high level abstraction, the Producer allows to connects to the topic in the cluster and to write (publish) messages on it. Under the hood, the library is managing connection pools, network buffering, waiting for brokers to acknowledge messages, retransmitting messages when necessary
- **Consumer** Consumers are the entities that process records/messages and can be configured to work independently on individual workloads or cooperatively with other consumers on a given workload (load balancing). Consumers manage how they process a workload based on their consumer group name: using a consumer group name allows consumers to be distributed within a single process, across multiple processes, and even across multiple systems.

2.5 Devices deployed for data retrieval

The whole system consists of the following devices:

- a **Raspberry Pi 3 model B** board, with 1GB of RAM and equipped with a 16GB microSD memory. It runs a Node-RED program, which is in charge of collecting data and producing the dataset, and it's connected to Internet through its in-built WiFi card and to an Arduino Uno board through a USB cable;
- an **Arduino Uno** board which receives commands from the Raspberry Pi for collection of data from sensors;
- an **MH-SENSOR-SERIES Flying Fish** light sensor (Fig. 1b), based on LM-393 comparator. It needs a 5V input voltage and can work either in digital or in analog;
- a **BME-280** environmental sensor (Fig. 1a), able to detect three different air information: temperature, humidity and pressure; it is very compact and consumes very few power, as it needs only $1mA$ when it's working and $5\mu A$ when it's in stand-by mode.

(a) *BME280 environmental sensor* ***(b)*** *Light sensor*

Figure 1: *Sensors used to gather data*

3 Analysis of supplied dataset with Weka

In our project, **Weka** has been used a lot to evaluate the data, in particular, the goal was to explore and apply various algorithms, starting from the pre-processing stage and continuing with clustering and classification, and in the end trying to exploit association rules.
First of all, we used a datasets provided by the professor, to better understand the tool and to compare the data provided to the data acquired by us, described in the chapter 4.

3.1 Datasets supplied structure and manipulation

The datasets supplied by the professor is composed by 19 csv file in which for each row there are some values related to environment in which the sensors has been put and following there are the values of each column:

- *timestamp*, indicates the exact moment of the row acquisition;
- *wind _speed_value*, expressed in m/s, expresses the value of the velocity of the speed;
- *wind_force*, defines the force of the wind using the same unit of measurements of the wind speed value (m/s);
- *wind_direction*, is a numerical value between 0 and 7, the direction of true north is 0, the value is increased clockwise, and the value of true east is 2;
- *wind_direction_degrees*, defines the direction of the wind in degrees where the values lies between 0° and 360°, the direction of true north is 0° and the degree increases clockwise, and the direction of true east is 90°;
- *humidity*, expresses the quality of the air in percentage number, from 0 (very dry) to 100 (very wet);
- *temperature*, defined in Celsius degree, is the environment temperature;
- *noise*, expressed in **Decibel**;
- *PM2.5* is a numerical classification given to particulate matter based on the average size of its particles. The term *PM2.5* groups together all particles with a size smaller than or equal to 2.5, and this values is expressed in $\mu g/m^3$ (micrograms per cubic meter);
- *PM10*, same as before, but now is considered particles with diameter no grater than 10 μm;
- *Atmospheric_pressure*, also known as barometric pressure (after the barometer), is the pressure within the atmosphere of Earth and it's expressed in **kPa**;
- *light*, which is referred to the luminosity, and the unity of measurements is **lux**
- *hundred_light*, same as before, but in this case is defined in percentage of **hundred lux**;
- *optical_rainfall*, represents the rainfalls and it's expressed in **mm**;

After a careful and accurate analysis of the data and the period of their acquisition, we realized that each day they were collected very frequently, with an average of 16 seconds between one acquisition and another, so it was decided to create a python script to facilitate the pre-processing work later. With the python script we were able to merge all the csv files of the dataset, mentioned before, in a single file in which in each row there is an acquisition; the only change that has been made respect the supplied dataset is related to the timestamp, in the merged file there isn't the *timestamp* value of the acquisition but there are two values related to the time: the *date* and *time_range*. The date is taken by the timestamp of the supplied dataset and the time range refers to a period of time of 15 minutes and the values of each row are the result of the average of the respective 15 minutes in the original dataset.

3.2 Pre-processing stage

Now that we have created a single file that's easier to work with, it's time to start analyzing and evaluating our data. Before starting to work with the data and apply the classification and clustering algorithms, a pre-processing phase is performed on the data. The pre-processing techniques help in cleaning and transforming the data to improve quality and relevance of the input for subsequent analysis.
First of all, we uploaded the file on Weka and initially we have all the values present in the dataset, as explained above.
The first operation done was to delete the superfluous values, not useful for our analyses. The values deleted are:

- *date*, because we considered that the values of the date such as year, month and day are useless for our purposes. We left the **time_range** and we evaluated this value for our goals;
- *wind_speed_value*;
- *wind_direction*;

- *wind_direction_degrees*, we decided to use only the **wind_force** in our analysis and we deleted these three values of wind because they are more similar and they resulted as if they were duplicates of the value that we decided not to eliminate;
- *noise*;
- *PM10*, we evaluated this one and PM2.5 and we have chosen to leave **PM2.5** for our purposes;
- *Atmosphere_pressure*;
- *hundred_light*;
- *optical_rainfall*;

After eliminating the previous values, considered obsolete for our objective, we have 5 values that we will discretize in a second phase, the next one. Weka provides option for transforming continuous numeric attributes into discrete ones. Discretization is particularly useful when dealing with algorithms that require categorical data or when users want to simplify the analysis by working with predefined intervals or bins.
The remaining values, and therefore discretized, are the following:

- **wind_force**, discretized in 4 bins;
- **humidity**, discretized in 4 bins;
- **temperature**, discretized in 4 bins;
- **PM2.5**, discretized in 3 bins;
- **light**, discretized in 5 bins and subsequently it's applied the function **MergeManyValues** to the last 4 bins in order to create only one bin. It was decided to proceed in this way, joining the last 4 bins, in such a way as to have two balanced bins in the end.

As concerns the **time_range**, we used also in this case the function **MargeManyValues** in order to have 12 classes of time that refer to time intervals of 2 hours each for a single day (Ex. 00-02, 02-04, and so on and so forth).
To conclude this phase, by analyzing the temperature and humidity values, we realize that humidity and temperature are correlated as we notice from the graphs that low humidity values are found almost exclusively at high temperatures; conversely, high humidity values are found in most cases at low temperatures.

3.3 Classification phase

After the pre-processing phase, the dataset was ready and we were able to apply the classification algorithms and to evaluate the data for our analysis.

3.3.1 Naive Bayes Classifier

Naive Bayes Classifier is a probabilistic machine learning algorithm that is widely used for classification tasks. It's based on Bayes' theorem with the assumption of independence between features. Based on the probabilities of a data instance's characteristics, the Naive Bayes classifier determines the probability that it belongs to a certain class. It is called "naive" because it assumes that all features are conditionally independent given the class. This assumption simplifies the computation and allows the classifier to work well even with limited training data. The algorithm calculates the prior probability of each class and the conditional probabilities of the features given the class in order to train a Naive Bayes classifier. The training data are used to determine there probabilities using maximum likelihood estimation or other techniques. By applying Bayes' theorem, the classifier utilizes these probabilities to create prediction on data.
We chosen to start with this algorithm because it is the simplest of all those that will be proposed and in the following paragraphs the application of it on the humidity and temperature present in the starting dataset will be explained.
First of all, the **Naive Bayes** algorithm has been applied to the humidity in its simplest form, not modifying any parameters in the Weka settings.
As we can see, in the Figure 2a, the class a is the least performing with a TP Rate equal to 0,524. To try to improve this result, the algorithm Cost Sensitive Classifier, explained subsequently was applied.
The Cost Sensitive Classifier in Weka allows to incorporate the costs of misclassification into the Naive Bayes classification process. It's developed to address situations where different misclassification errors have varying costs or where the cost of misclassifying different classes is imbalanced. In traditional classification we know that each error is considered equally costly. The Cost Sensitive Classifier enables to assign a specific costs to different types of misclassification errors. The algorithm, rather than focusing just on accuracy, tries to reduce the overall cost of misclassification by taking costs into account during the classification process.

So, at this point, the algorithm described previously has been applied to the Naive Bayes algorithm. In the cost matrix, as you can see in the following recap, the cost for class b has been changed trying to improve class a.

```
Time taken to build model: 0.01 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1177               63.2456 %
Incorrectly Classified Instances       684               36.7544 %
Kappa statistic                          0.4733
Mean absolute error                      0.2224
Root mean squared error                  0.3519
Relative absolute error                 63.6219 %
Root relative squared error             84.1659 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,524    0,014    0,786      0,524    0,629      0,614    0,933     0,647     '(-inf-50.05]'
               0,585    0,145    0,500      0,585    0,539      0,416    0,833     0,534     '(50.05-64.6]'
               0,637    0,261    0,559      0,637    0,595      0,365    0,773     0,583     '(64.6-79.15]'
               0,680    0,106    0,791      0,680    0,731      0,595    0,867     0,815     '(79.15-inf)'
Weighted Avg.  0,632    0,158    0,653      0,632    0,637      0,483    0,834     0,665

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
  88  53  25   2 |   a = '(-inf-50.05]'
  24 216 107  22 |   b = '(50.05-64.6]'
   0 131 405 100 |   c = '(64.6-79.15]'
   0  32 188 468 |   d = '(79.15-inf)'
```

(a) *Humidity Classic Naive Bayes*

```
Cost Matrix
 0    1.8  1    1
 1    0    1    1
 1    1    0    1
 1    1    1    0


Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1179               63.353  %
Incorrectly Classified Instances       682               36.647  %
Kappa statistic                          0.4748
Mean absolute error                      0.2233
Root mean squared error                  0.3518
Relative absolute error                 63.8758 %
Root relative squared error             84.1632 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,530    0,014    0,788      0,530    0,633      0,619    0,934     0,647     '(-inf-50.05]'
               0,585    0,143    0,502      0,585    0,541      0,418    0,831     0,533     '(50.05-64.6]'
               0,638    0,261    0,559      0,638    0,596      0,367    0,773     0,583     '(64.6-79.15]'
               0,680    0,106    0,791      0,680    0,731      0,595    0,866     0,814     '(79.15-inf)'
Weighted Avg.  0,634    0,158    0,654      0,634    0,638      0,484    0,834     0,664

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
  89  52  25   2 |   a = '(-inf-50.05]'
  24 216 107  22 |   b = '(50.05-64.6]'
   0 130 406 100 |   c = '(64.6-79.15]'
   0  32 188 468 |   d = '(79.15-inf)'
```

(b) *Humidity Cost Sensitive Naive Bayes*

Figure 2: *Naive Bayes on Humidity Values*

In conclusion, it is possible to assert that there has been a slight improvement for class a, however, it always remains the one that performs less than all the others.
After analyzed the Naive Bayes algorithm on the Humidity values, now it's the time to evaluate this algorithm on the Temperature values. Initially it was run without modifying the Weka settings and in the Figure **??** it's possible to see the results.
As we can see in the Figure 3a, the value of *Correctly Classified Instance* is equal to 71.7356%.
We immediately realized that by using the Naive Bayes classifier we got good results right from the start. We can try to improve even more by using the Cost Sensitive here too, as we did previously. To apply this algorithm, therefore, we will work on the weights of the matrix, in particular we will modify the weight relating to the b in the row of the c to be able to improve the TP Rate, as it is the lowest, the only one below 0.6%.

```
Time taken to build model: 0.01 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1335               71.7356 %
Incorrectly Classified Instances       526               28.2644 %
Kappa statistic                          0.5623
Mean absolute error                      0.1818
Root mean squared error                  0.3201
Relative absolute error                 55.9663 %
Root relative squared error             79.4515 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,789    0,062    0,738      0,789    0,763      0,709    0,941     0,796     '(-inf-10.575]'
               0,763    0,249    0,749      0,763    0,756      0,513    0,835     0,802     '(10.575-15.85]'
               0,595    0,135    0,615      0,595    0,605      0,465    0,824     0,635     '(15.85-21.125]'
               0,670    0,007    0,852      0,670    0,750      0,742    0,977     0,775     '(21.125-inf)'
Weighted Avg.  0,717    0,170    0,718      0,717    0,717      0,550    0,860     0,755

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 265  65   6   0 |   a = '(-inf-10.575]'
  74 701 144   0 |   b = '(10.575-15.85]'
  20 167 294  13 |   c = '(15.85-21.125]'
   0   3  34  75 |   d = '(21.125-inf)'
```

(a) *Temperature Classic Naive Bayes*

```
Cost Matrix
 0 1 1 1
 1 0 1 1
 1 2 0 1
 1 1 1 0


Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1333               71.6282 %
Incorrectly Classified Instances       528               28.3718 %
Kappa statistic                          0.566
Mean absolute error                      0.1847
Root mean squared error                  0.3232
Relative absolute error                 56.8659 %
Root relative squared error             80.213  %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,789    0,062    0,738      0,789    0,763      0,709    0,940     0,794     '(-inf-10.575]'
               0,728    0,212    0,770      0,728    0,748      0,517    0,833     0,801     '(10.575-15.85]'
               0,658    0,162    0,594      0,658    0,624      0,480    0,824     0,635     '(15.85-21.125]'
               0,661    0,007    0,860      0,661    0,747      0,741    0,976     0,776     '(21.125-inf)'
Weighted Avg.  0,716    0,159    0,723      0,716    0,718      0,555    0,859     0,754

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 265  60  11   0 |   a = '(-inf-10.575]'
  74 669 176   0 |   b = '(10.575-15.85]'
  20 137 325  12 |   c = '(15.85-21.125]'
   0   3  35  74 |   d = '(21.125-inf)'
```

(b) *Temperature Cost Sensitive Naive Bayes*

Figure 3: *Naive Bayes on Temperature Values*

3.3.2 K Nearest Neighbors

The K-Nearest Neighbors (KNN) algorithm is a type of instance-based learning, which means that it doesn't learn a model. Instead, it uses the entire training dataset to make predictions. In KNN, we classify a new instance by looking at the K nearest instances from the training dataset, and choosing the majority class among those K nearest neighbors.
To use KNN for classification, we need to choose a value for K. A small value of K can lead to overfitting, while a large value of K can lead to underfitting. Therefore, we need to choose an appropriate value of K that balances between overfitting and underfitting. We can do this by using cross-validation.
KNN is a computationally expensive algorithm, especially when dealing with large datasets. Therefore, it's important to preprocess the data to reduce the number of features and remove any noise or outliers.
First of all, the KNN algorithm was applied to the values of humidity. Through the evaluation of different k we arrived at the conclusion that the best performing value of k is 1. This is also indicated by the value of *crossValidate* set to TRUE. In this case this value finds the best k between 1 and the k chosen by us in the algorithm characteristics table. In the Figure 4 are shown the results of the first application of the algorithm, in particular the **linear KNN**.
Subsequently, in order to be able to evaluate the execution performances, we modify the search algorithms to understand if it is possible to find a better solution than the one already found previously. We realized that by modifying the search algorithms, no improvement of the algorithm is achieved compared to the vanilla test.
So, now let's try to do some tests keeping the linear KNN used in the first part as the search algorithm, but we are going to modify the distance algorithm, the default one was the Euclidean distance. In the following table it's possible to see the different results and the values of correctly classified instances with the different distance algorithms.

Distance Algorithm	Correctly Classified Instances	Info
Euclidian Distance	64.16%	
Chebyshev Distance	64.21%	
Filtered Distance	64.32%	Very slow
Manhattan Distance	64.16%	
Minkowski Distance	64.16%	

Table 1: *Result's comparison distance algorithms*

As we can see table 1, the best result, even if with a very minimal difference, was obtained using the *Filtered Distance* and in the Figure 5 it's possible to see all results of the algorithm with this distance.

```
=== Classifier model (full training set) ===

IB1 instance-based classifier
using 1 nearest neighbour(s) for classification


Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1194               64.1591 %
Incorrectly Classified Instances       667               35.8409 %
Kappa statistic                          0.4866
Mean absolute error                      0.2063
Root mean squared error                  0.3332
Relative absolute error                 59.0185 %
Root relative squared error             79.6997 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,595    0,050    0,543      0,595    0,568      0,524    0,950     0,719     '(-inf-50.05]'
               0,472    0,092    0,558      0,472    0,511      0,405    0,865     0,569     '(50.05-64.6]'
               0,700    0,259    0,584      0,700    0,637      0,425    0,816     0,653     '(64.6-79.15]'
               0,690    0,109    0,788      0,690    0,736      0,600    0,888     0,855     '(79.15-inf)'
Weighted Avg.  0,642    0,152    0,650      0,642    0,642      0,494    0,864     0,717

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 100  46  20   2 |   a = '(-inf-50.05]'
  61 174 112  22 |   b = '(50.05-64.6]'
  21  66 445 104 |   c = '(64.6-79.15]'
   2  26 185 475 |   d = '(79.15-inf)'
```

Figure 4: *Humidity Linear KNN*

```
=== Classifier model (full training set) ===

IB1 instance-based classifier
using 1 nearest neighbour(s) for classification


Time taken to build model: 0.03 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1197               64.3203 %
Incorrectly Classified Instances       664               35.6797 %
Kappa statistic                          0.489
Mean absolute error                      0.2055
Root mean squared error                  0.3341
Relative absolute error                 58.7694 %
Root relative squared error             79.9231 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,607    0,050    0,545      0,607    0,575      0,531    0,951     0,724     '(-inf-50.05]'
               0,472    0,090    0,563      0,472    0,513      0,408    0,863     0,571     '(50.05-64.6]'
               0,701    0,258    0,585      0,701    0,638      0,428    0,816     0,643     '(64.6-79.15]'
               0,690    0,109    0,788      0,690    0,736      0,600    0,885     0,853     '(79.15-inf)'
Weighted Avg.  0,643    0,151    0,652      0,643    0,644      0,497    0,863     0,714

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 102  45  19   2 |   a = '(-inf-50.05]'
  62 174 111  22 |   b = '(50.05-64.6]'
  21  65 446 104 |   c = '(64.6-79.15]'
   2  25 186 475 |   d = '(79.15-inf)'
```

Figure 5: *Linear KNN on Humidity-Filtered Distance*

As made with the previously algorithm, also in this case, we evaluated the Cost Sensitive Classifier again to see if it was possible to improve the algorithm previously performed. We have noticed, as we expected, that in this case through the use of this algorithm there is no improvement such as to be able to take it into consideration. As the class b improves, but the class a worsens in a greater proportion than class b improves. Since the purpose of cost sensitive is to obtain good TP Rates for each class by improving the results of the worst class, even at the cost of obtaining generally worse results. In this case, the worst situation (which we wanted to improve) is moved from one class to another, so no benefit was found in using it.

```
Cost Matrix
 0    1    1    1
 1    0    1.5  1
 1    1    0    1
 1    1    1    0


Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1196               64.2665 %
Incorrectly Classified Instances       665               35.7335 %
Kappa statistic                          0.4864
Mean absolute error                      0.2056
Root mean squared error                  0.3344
Relative absolute error                 58.8226 %
Root relative squared error             79.9797 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,470    0,031    0,603      0,470    0,528      0,492    0,950     0,721     '(-inf-50.05]'
               0,550    0,115    0,543      0,550    0,546      0,433    0,863     0,570     '(50.05-64.6]'
               0,690    0,256    0,583      0,690    0,632      0,419    0,816     0,645     '(64.6-79.15]'
               0,690    0,109    0,788      0,690    0,736      0,600    0,885     0,854     '(79.15-inf)'
Weighted Avg.  0,643    0,153    0,653      0,643    0,644      0,495    0,863     0,714

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
  79  68  19   2 |   a = '(-inf-50.05]'
  35 203 109  22 |   b = '(50.05-64.6]'
  15  78 439 104 |   c = '(64.6-79.15]'
   2  25 186 475 |   d = '(79.15-inf)'
```

Figure 6: *Cost Sensitive KNN on Humidity*

We ran the KNN algorithm also to the Temperature values to evaluate the performances. From Weka the best k to use was the 1, but we decided to continue the evaluation of performance with k equal to 3. The advantages of this decision are essentially two: the robustness to outliers, in fact considering multiple nearest neighbors helps to mitigate the influence of outliers that are less likely to affect the predictions as the algorithm takes into account the collective opinion of the three closest neighbors, the second advantages is that the overfitting is reduced because considering more neighbors, the model generalizes better to unseen data. In the Figure 7a it's possible to see all the results on Weka. Subsequently, to find an even better result, different search algorithms have been

evaluated and in the table 2 there is the comparison between them.

K	Search Algorithm	Correctly Classified Instances
1	Linear NN	76.78%
1	Ball Tree	76.78%
1	Cover Tree	76.78%
1	Filtered Neighbor	76.78%
1	KD Tree	77%
3	Linear NN	75.68%
3	Ball Tree	75.66%
3	Cover Tree	75.65%
3	Filtered Neighbor	75.65%
3	KD Tree	76.41%

Table 2: *Search algorithm comparison*

Also in this case we will evaluate whether it is possible to have even better results by applying the Cost Sensitive meta-algorithm. As explained previously, also in this case the value of k taken into account is 3. By applying this algorithm, we realized that there are no improvements over running the standard algorithm, as can be seen in the Figure 7b

```
IB1 instance-based classifier
using 3 nearest neighbour(s) for classification


Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1406               75.6582 %
Incorrectly Classified Instances       453               24.3418 %
Kappa statistic                          0.6186
Mean absolute error                      0.1646
Root mean squared error                  0.297
Relative absolute error                 50.6789 %
Root relative squared error             73.7169 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure  MCC      ROC Area  PRC Area  Class
               0,860     0,057     0,769      0,860     0,812      0,769    0,965     0,873     '(-inf-10.575]'
               0,838     0,254     0,763      0,838     0,799      0,586    0,869     0,858     '(10.575-15.85]'
               0,547     0,083     0,705      0,547     0,616      0,507    0,848     0,678     '(15.85-21.125]'
               0,705     0,008     0,849      0,705     0,771      0,761    0,964     0,789     '(21.125-inf)'
Weighted Avg.  0,757     0,158     0,754      0,757     0,751      0,609    0,886     0,809

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 289  47   0   0 |   a = '(-inf-10.575]'
  65 770  83   1 |   b = '(10.575-15.85]'
  22 189 270  13 |   c = '(15.85-21.125]'
   0   3  30  79 |   d = '(21.125-inf)'
```

(a) *K Nearest Neighbors on Temperature*

```
Cost Matrix
 0 1 1 1
 1 0 1 1
 1 3 0 1
 1 1 1 0


Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1382               74.2611 %
Incorrectly Classified Instances       479               25.7389 %
Kappa statistic                          0.6025
Mean absolute error                      0.1688
Root mean squared error                  0.3001
Relative absolute error                 51.9574 %
Root relative squared error             74.4812 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure  MCC      ROC Area  PRC Area  Class
               0,845     0,047     0,798      0,845     0,821      0,780    0,966     0,875     '(-inf-10.575]'
               0,785     0,210     0,785      0,785     0,785      0,574    0,871     0,861     '(10.575-15.85]'
               0,617     0,143     0,610      0,617     0,614      0,473    0,848     0,678     '(15.85-21.125]'
               0,643     0,008     0,837      0,643     0,727      0,719    0,960     0,784     '(21.125-inf)'
Weighted Avg.  0,743     0,151     0,744      0,743     0,742      0,593    0,888     0,811

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 284  40  12   0 |   a = '(-inf-10.575]'
  53 721 144   1 |   b = '(10.575-15.85]'
  19 157 305  13 |   c = '(15.85-21.125]'
   0   1  39  72 |   d = '(21.125-inf)'
```

(b) *K Nearest Neighbors Cost Sensitive on Temperature*

Figure 7: *K Nearest Neighbors on Temperature Values*

3.3.3 Random Forest

Random Forest is a type of ensemble learning algorithm that combines multiple decision trees to improve predictions. Each individual decision tree in the forest is trained on a subset of the data and uses a random subset of features. To make a prediction, the forest aggregates the predictions of all the individual trees.
Random Forest is a popular algorithm for classification and regression problems, and it has several advantages over other methods. It can handle large datasets with many features and can handle missing data. It also provides a measure of feature importance, which can be useful for understanding the underlying relationships between features and the target variable.
Overall, Random Forest is a powerful and flexible algorithm that can be used for a wide range of classification and regression problems. Now, to do a comparison between classification algorithms it's time to apply the Random Forest, described previously, to the humidity values and, in the first instance, we did this not modifying the Weka settings. The results are shown in the Figure 8a.
After we ran the Cost Sensitive for the Random Forest, in order to find a best results and improvements in the analysis. Comparing the normal algorithm and the Cost Sensitive we realized that in principle the *Correctly Classified Instances* value improves very little up to 65.29%, as shown in Figure 9b however we believe that it is not a value to take into consideration as class a is a very sensitive class and consequently with this algorithm the TP Rate drops drastically and is equal to 0.440 compared to the previous 0.560 of initial test.

```
RandomForest

Bagging with 100 iterations and base learner

weka.classifiers.trees.RandomTree -K 0 -M 1.0 -V 0.001 -S 1 -do-not-check-capabilities

Time taken to build model: 0.2 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1209               64.9651 %
Incorrectly Classified Instances       652               35.0349 %
Kappa statistic                          0.4968
Mean absolute error                      0.2068
Root mean squared error                  0.3315
Relative absolute error                 59.166  %
Root relative squared error             79.2979 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,560    0,041    0,577      0,560    0,568      0,526    0,954     0,726     '(-inf-50.05]'
               0,501    0,095    0,566      0,501    0,532      0,426    0,864     0,577     '(50.05-64.6]'
               0,698    0,254    0,588      0,698    0,638      0,429    0,818     0,649     '(64.6-79.15]'
               0,706    0,111    0,789      0,706    0,745      0,611    0,891     0,858     '(79.15-inf)'
Weighted Avg.  0,650    0,150    0,657      0,650    0,650      0,504    0,866     0,719

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
  94  53  19   2 |   a = '(-inf-50.05]'
  50 185 112  22 |   b = '(50.05-64.6]'
  19  67 444 106 |   c = '(64.6-79.15]'
   0  22 180 486 |   d = '(79.15-inf)'
```

(a) *Random Forest on Humidity*

```
Cost Matrix
 0 1 1 1
 1 0 2 1
 1 1 0 1
 1 1 1 0


Time taken to build model: 0.09 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1215               65.2875 %
Incorrectly Classified Instances       646               34.7125 %
Kappa statistic                          0.5007
Mean absolute error                      0.2075
Root mean squared error                  0.3325
Relative absolute error                 59.3578 %
Root relative squared error             79.5355 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,440    0,019    0,692      0,440    0,538      0,518    0,952     0,717     '(-inf-50.05]'
               0,618    0,123    0,553      0,618    0,584      0,475    0,864     0,575     '(50.05-64.6]'
               0,676    0,243    0,591      0,676    0,630      0,421    0,818     0,652     '(64.6-79.15]'
               0,702    0,112    0,787      0,702    0,742      0,606    0,891     0,858     '(79.15-inf)'
Weighted Avg.  0,653    0,151    0,665      0,653    0,654      0,509    0,866     0,719

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
  74  74  18   2 |   a = '(-inf-50.05]'
  20 228  99  22 |   b = '(50.05-64.6]'
  13  86 430 107 |   c = '(64.6-79.15]'
   0  24 181 483 |   d = '(79.15-inf)'
```

(b) *Cost Sensitive Random Forest on Humidity*

Figure 8: *Random Forest on Humidity Values*

The same was done to the Temperature Values. In fact, first of all, we applied the Random Forest algorithm without changing any parameters in Weka, and the results can be seen in the Figure 9a.
At a later time, in order to improve the results of the previous execution, we ran the Cost Sensitive in the Random Forest algorithm to the Temperature values. Also in this case, class C is the one that performs worse than the

others, therefore, we will work on the weight relating to this class in the cost matrix. By modifying the weight in the cell of row C and column B, setting it to 2, we obtain 76.41% of Correctly Classified Instances. We therefore realize that it is the best result obtainable with the use of this algorithm.

```
=== Classifier model (full training set) ===

RandomForest

Bagging with 100 iterations and base learner

weka.classifiers.trees.RandomTree -K 0 -M 1.0 -V 0.001 -S 1 -do-not-check-capabilities

Time taken to build model: 0.25 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1433               77.0016 %
Incorrectly Classified Instances       428               22.9984 %
Kappa statistic                          0.6402
Mean absolute error                      0.1581
Root mean squared error                  0.2917
Relative absolute error                 48.6695 %
Root relative squared error             72.3941 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,870    0,050    0,793      0,870    0,830      0,796    0,964     0,876     '(-inf-10.575]'
               0,842    0,239    0,775      0,842    0,807      0,605    0,873     0,856     '(10.575-15.85]'
               0,573    0,083    0,713      0,573    0,635      0,528    0,858     0,696     '(15.85-21.125]'
               0,723    0,007    0,871      0,723    0,790      0,782    0,951     0,808     '(21.125-inf)'
Weighted Avg.  0,770    0,150    0,767      0,770    0,765      0,630    0,890     0,814

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 295  41   0   0 |   a = '(-inf-10.575]'
  58 774  85   2 |   b = '(10.575-15.85]'
  19 182 283  10 |   c = '(15.85-21.125]'
   0   2  29  81 |   d = '(21.125-inf)'
```

(a) *Random Forest on Temperature*

```
Bagging with 100 iterations and base learner

weka.classifiers.trees.RandomTree -K 0 -M 1.0 -V 0.001 -S 1 -do-not-check-capabilities

Cost Matrix
 0 1 1 1
 1 0 1 1
 1 2 0 1
 1 1 1 0


Time taken to build model: 0.13 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1422               76.4105 %
Incorrectly Classified Instances       439               23.5895 %
Kappa statistic                          0.6332
Mean absolute error                      0.1611
Root mean squared error                  0.2936
Relative absolute error                 49.5959 %
Root relative squared error             72.8546 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,872    0,049    0,798      0,872    0,834      0,796    0,964     0,876     '(-inf-10.575]'
               0,820    0,219    0,785      0,820    0,803      0,602    0,873     0,856     '(10.575-15.85]'
               0,607    0,108    0,671      0,607    0,638      0,517    0,857     0,697     '(15.85-21.125]'
               0,670    0,007    0,862      0,670    0,754      0,747    0,951     0,804     '(21.125-inf)'
Weighted Avg.  0,764    0,146    0,762      0,764    0,761      0,623    0,890     0,814

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 293  39   4   0 |   a = '(-inf-10.575]'
  55 754 108   2 |   b = '(10.575-15.85]'
  19 165 300  10 |   c = '(15.85-21.125]'
   0   2  35  75 |   d = '(21.125-inf)'
```

(b) *Cost Sensitive Random Forest on Temperature*

Figure 9: *Random Forest on Temperature Values*

3.3.4 K Star

The K-Star algorithm is a type of instance-based learning algorithm that uses the entire training dataset to make predictions. Like KNN, it classifies a new instance by looking at the K nearest instances from the training dataset, and choosing the majority class among those K nearest neighbors. However, K-Star also uses a weighted voting scheme that takes into account the similarity between the new instance and each of the K nearest neighbors. The weights are based on the distance between the new instance and each of the K nearest neighbors, and they are used to adjust the contribution of each neighbor to the final classification.
To use K-Star for classification, we need to choose a value for K. As with KNN, a small value of K can lead to overfitting, while a large value of K can lead to underfitting. Therefore, we need to choose an appropriate value of K that balances between overfitting and underfitting. We can do this by using cross-validation.
K-Star is a computationally expensive algorithm, especially when dealing with large datasets. Therefore, it's important to preprocess the data to reduce the number of features and remove any noise or outliers.
Overall, K-Star can be a good choice for classification problems when we have a small to medium-sized dataset with a low number of features, and when the classes are well-separated.
First of all, we applied the K-Star algorithm without modifying the Weka settings and in the Figure 10 there are all the results. Through the use of the K-Star we have the best result compared to all the algorithms seen previously. To try to improve this result even more let's evaluate other Blend values.
The best blend value is 15. In the Figure 11 we see that the correctly classified instances is equal to 66.79%,

improving slightly compared to the previous one.
By analyzing the various classes seen previously, we realize that the TP Rate for class a is relatively low and therefore in this phase we will analyze the various weights of the matrix to be able to improve this value.
By applying the cost sensitive to the k-star, and in the Figure 12 there are the results, we are able to ensure that each class has an acceptable value of TP Rate, even if in some classes this worsens slightly compared to the algorithm seen previously.

```
KStar options : -B 20 -M a

Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1238               66.5234 %
Incorrectly Classified Instances       623               33.4766 %
Kappa statistic                          0.5131
Mean absolute error                      0.2424
Root mean squared error                  0.337
Relative absolute error                 69.3258 %
Root relative squared error             80.602  %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,417    0,012    0,778      0,417    0,543      0,541    0,954     0,710     '(-inf-50.05]'
               0,528    0,094    0,582      0,528    0,554      0,451    0,863     0,584     '(50.05-64.6]'
               0,783    0,296    0,578      0,783    0,665      0,463    0,809     0,640     '(64.6-79.15]'
               0,690    0,085    0,826      0,690    0,752      0,632    0,882     0,849     '(79.15-inf)'
Weighted Avg.  0,665    0,152    0,689      0,665    0,664      0,530    0,860     0,712

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
  70  68  27   3 |   a = '(-inf-50.05]'
  20 195 142  12 |   b = '(50.05-64.6]'
   0  53 498  85 |   c = '(64.6-79.15]'
   0  19 194 475 |   d = '(79.15-inf)'
```

Figure 10: *K Star on Humidity*

```
KStar Beta Verion (0.1b).
Copyright (c) 1995-97 by Len Trigg (trigg@cs.waikato.ac.nz).
Java port to Weka by Abdelaziz Mahoui (aml4@cs.waikato.ac.nz).

KStar options : -B 15 -M a

Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1243               66.792  %
Incorrectly Classified Instances       618               33.208  %
Kappa statistic                          0.5176
Mean absolute error                      0.2362
Root mean squared error                  0.3345
Relative absolute error                 67.559  %
Root relative squared error             80.009  %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,429    0,012    0,783      0,429    0,554      0,551    0,955     0,720     '(-inf-50.05]'
               0,539    0,095    0,584      0,539    0,561      0,458    0,867     0,588     '(50.05-64.6]'
               0,781    0,291    0,583      0,781    0,668      0,467    0,812     0,644     '(64.6-79.15]'
               0,690    0,085    0,826      0,690    0,752      0,632    0,884     0,852     '(79.15-inf)'
Weighted Avg.  0,668    0,151    0,691      0,668    0,667      0,534    0,862     0,717

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
  72  69  24   3 |   a = '(-inf-50.05]'
  20 199 138  12 |   b = '(50.05-64.6]'
   0  54 497  85 |   c = '(64.6-79.15]'
   0  19 194 475 |   d = '(79.15-inf)'
```

Figure 11: *K Star on Humidity Best Blend*

```
Cost Matrix
 0    2    1    1
 1    0    1.5  1
 1    1    0    1
 1    1    1    0


Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1193               64.1053 %
Incorrectly Classified Instances       668               35.8947 %
Kappa statistic                          0.4834
Mean absolute error                      0.2392
Root mean squared error                  0.3393
Relative absolute error                 68.4145 %
Root relative squared error             81.1672 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0,524    0,032    0,620      0,524    0,568      0,531    0,947     0,695     '(-inf-50.05]'
                 0,518    0,099    0,565      0,518    0,540      0,433    0,861     0,581     '(50.05-64.6]'
                 0,715    0,287    0,565      0,715    0,631      0,411    0,800     0,639     '(64.6-79.15]'
                 0,667    0,099    0,798      0,667    0,727      0,594    0,878     0,842     '(79.15-inf)'
Weighted Avg.    0,641    0,157    0,656      0,641    0,643      0,494    0,854     0,707

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
  88  56  21   3 |   a = '(-inf-50.05]'
  39 191 125  14 |   b = '(50.05-64.6]'
  14  68 455  99 |   c = '(64.6-79.15]'
   1  23 205 459 |   d = '(79.15-inf)'
```

Figure 12: *Cost Sensitive K Star on Humidity*

As we done with the other algorithms, also in this case, we are going to apply the **K Star** algorithm to the Temperature values in order to analysis these parameters too.
By evaluating different values of Blend, unlike the analyzes conducted in humidity, we realize that in this case as the Blend value decreases, the Correctly Classified Instances value increases. The best value obtained is with Blend=3 and we have a Correctly Classified Instances of 77.32%. (Figure 13)
Trying to improve the value of Correctly Classified Instances, we apply the Cost Sensitive algorithm and, as previously done, the weight of class C in matrix costs of Figure 12 is modified since it is the one that performs less than the others.

```
=== Classifier model (full training set) ===

KStar Beta Verion (0.1b).
Copyright (c) 1995-97 by Len Trigg (trigg@cs.waikato.ac.nz).
Java port to Weka by Abdelaziz Mahoui (aml4@cs.waikato.ac.nz).

KStar options : -B 3 -M a

Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1439               77.324  %
Incorrectly Classified Instances       422               22.676  %
Kappa statistic                          0.6439
Mean absolute error                      0.1697
Root mean squared error                  0.2904
Relative absolute error                 52.2424 %
Root relative squared error             72.0746 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0,869    0,047    0,804      0,869    0,835      0,798    0,970     0,883     '(-inf-10.575]'
                 0,854    0,244    0,773      0,854    0,812      0,613    0,875     0,863     '(10.575-15.85]'
                 0,571    0,080    0,721      0,571    0,637      0,532    0,862     0,700     '(15.85-21.125]'
                 0,714    0,007    0,870      0,714    0,784      0,776    0,975     0,827     '(21.125-inf)'
Weighted Avg.    0,773    0,151    0,771      0,773    0,768      0,635    0,895     0,821

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 292  44   0   0 |   a = '(-inf-10.575]'
  52 785  80   2 |   b = '(10.575-15.85]'
  19 183 282  10 |   c = '(15.85-21.125]'
   0   3  29  80 |   d = '(21.125-inf)'
```

Figure 13: *K Star on Temperature*

```
KStar options : -B 3 -M a

Cost Matrix
 0    1    1    1
 1    0    1    1
 1    1.5  0    1
 1    1    1    0


Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        1391               74.7448 %
Incorrectly Classified Instances       470               25.2552 %
Kappa statistic                          0.6076
Mean absolute error                      0.1739
Root mean squared error                  0.2983
Relative absolute error                 53.5215 %
Root relative squared error             74.0383 %
Total Number of Instances             1861

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0,845    0,049    0,793      0,845    0,818      0,777    0,967     0,875     '(-inf-10.575]'
               0,802    0,228    0,774      0,802    0,788      0,574    0,867     0,855     '(10.575-15.85]'
               0,601    0,123    0,639      0,601    0,619      0,488    0,854     0,677     '(15.85-21.125]'
               0,652    0,007    0,849      0,652    0,737      0,730    0,975     0,806     '(21.125-inf)'
Weighted Avg.  0,747    0,155    0,746      0,747    0,746      0,597    0,888     0,808

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 284  47   5   0 |   a = '(-inf-10.575]'
  52 737 128   2 |   b = '(10.575-15.85]'
  22 164 297  11 |   c = '(15.85-21.125]'
   0   4  35  73 |   d = '(21.125-inf)'
```

Figure 14: *Cost Sensitive K Star on Temperature*

A brief recap of all results obtained with different classification algorithms is shown in the Table 3. So, when we refer to the best balanced result, we mean that in the result of the algorithm, all the classes have a **TP Rate > 0.5**.

Type	Algorithm	Best overall result	Best balanced result
Humidity	Naive Bayes	63.35 %	63.35 %
	KNN	64.32 %	No results obtained
	Random Forest	65.29 %	64.96 %
	K-Star	66.79 %	64.10 %
Temperature	Naive Bayes	71.74 %	71.74 %
	KNN	75.66 % (K=3)	75.66 %
	Random Forest	77.00 %	77.00 %
	K-Star	77.32 %	77.32%

Table 3: *Results of Classification Algorithms*

3.4 Clustering phase

To do the best possible analysis on the given dataset, besides applying the classification algorithms, it is necessary to evaluate and apply the clustering algorithms, and this will be done in this section.
Clustering is a technique used in unsupervised machine learning to identify groups or clusters within a dataset. It partitions a set of data points into subsets or cluster based on their similarities or patterns. The clustering's aim is to group similar data points together while keeping dissimilar points in separate clusters.
The algorithm in clustering allocates data points to a cluster depending on how close or similar they are to one another. The choice of similarity measure depends on the specific cluster algorithm used.
The target variables or specified labels are not used in clustering techniques. Instead, they identify clusters by looking at the data's structure and connections. These clusters may highlight underlying patterns, similarities or subgroups that can be hidden in the dataset.

3.4.1 K Means

The **K Means** algorithm is a type of unsupervised machine learning algorithm used for clustering data. The algorithm works by dividing a set of data points into k groups or clusters based on their similarity. It starts by randomly selecting k centroids (one for each cluster) and then iterating through a process of assigning each data point to its closest centroid, and then updating the centroids based on the mean of the assigned data points. This process is repeated until the centroids no longer change or a maximum number of iterations is reached. It's important to highlight that this algorithm tries to minimize the sum of squared distance between each data point and its assigned centroid, which is known as the within-cluster sum of squares (SSE).
As regards our analysis and to be able to apply the algorithm described, we have the following attributes available, not discretized:

- **time_range**
- **wind_force**
- **humidity**
- **temperature**
- **PM2.5**

First of all, to run the algorithm, we ignored the *time_range* and *wind_force* attributes. After, we divided the dataset in two parts: the first one, 2/3 used for training set, and the second one 1/3 used for testing set We did a lot of proofs with different number of clusters from k=1 to k=8, in order to find a best solution and all the results are shown in the Table 4.

Clusters (K)	Training Set			Test Set		
	SSE	Decrease	Iterations	SSE	Decrease	Iterations
2	98.77	-	6	67.7	-	13
3	66.99	31,78	53	45.37	22,33	43
4	49.43	17,56	32	33.3	12,07	16
5	39.13	10,3	23	26.17	7,13	35
6	34.42	4,71	27	23.07	3,1	22
7	30.08	4,32	73	20.14	2,93	21
8	27.67	2,41	33	18.46	1,68	16

Table 4: *K-Means Comparison with different K parameters*

One way to determine the optimal number of clusters for K Means is by using the "elbow method" or "knee rule". This involves plotting the within-cluster sum of squares (SSE) for different values of k and selecting the value of k where the decrease in SSE begins to level off.
The "elbow" or "knee" in the plot represents the optimal value of k. In the case of this dataset, the elbow occurs at k=5, which is consistent with the results obtained in the previous analysis and in the Figure 15 it's possible to see the graph of this method.

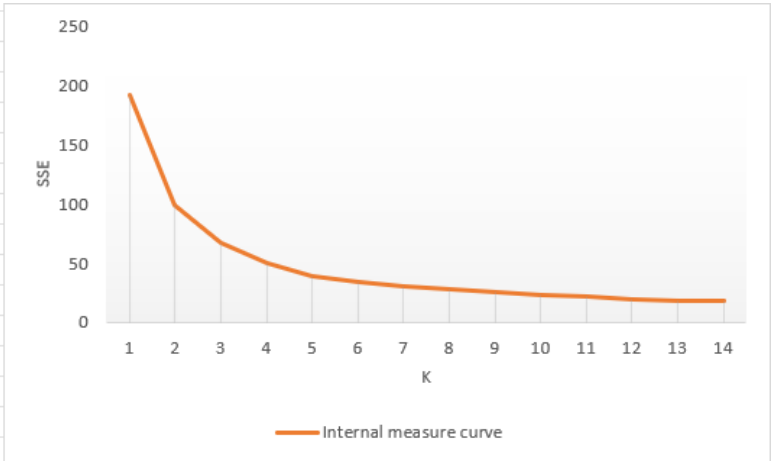
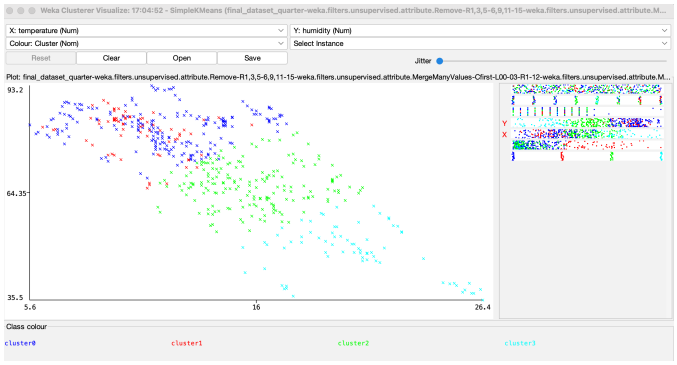


Figure 15: *Internal Measure Curve*

As said before, we obtained the best results for k=5. The following Figures, 16a and 16b, show the clusters obtained for k=4 and k=6. For k=4, hypothetically we thought that cluster 0 (blue) and cluster 1 (red) could be merged to obtain a well-defined cluster, as well as clusters 2 (green), 3 (light blue), and 4 (pink). For k=6, we evaluated that cluster 6 (yellow) and cluster 1 (red) could be merged as well as cluster 0 (blue) and cluster 5 (fuchsia) to obtain 5 well-defined clusters.



(a) *K Means Clustering with K=4*

(b) *K Means Clustering with K=6*

Figure 16: *K Means Clustering with K=4 and K=6*

3.5 Association Rules

Association rules are used to find relationship between different items in a dataset. These rules are defined as "if-then" statements, where the "if" part is a set of items and "then" part is another item that is often associated with the first set. The **support** and **confidence** values of each rule are used to determine how strong it is. Support refers to the percentage of transactions that contain both the "if" and "then" parts of the rule. Confidence refers to the percentage of transactions that contain the "if" part of the rule and the "then" part of the rule. High support and confidence values indicate strong associations between the items in the rule.
Weka provides the **Apriori** and **FP-Growth** algorithms to find the association rules of the items in the dataset. The two algorithms have been used to find association rules between different weather parameters such as humidity, temperature, wind force, light and PM2.5.
By analyzing the generated association rules, we can understand which weather parameters tend to occur together and how strongly they are associated with each other.
For example, the rule "*15.85<temperature<21.12* **=>** *PM2.5 < 56.43*" has a high confidence value of **0.96**, that comes out through running both algorithms, indicates that when temperature falls in the range (15.85-21.12), there is a strong likelihood that the PM2.5 level will be less than 56.43. Similarly, other rules have been generated and their strength has been evaluated based on support and confidence value.
All the rules found with the two algorithms are shown in the following Table 5

Association Rule		Algorithm		Confidence
Left Part	Right Part	Apriori	FP-Growth	
15.85 < temperature < 21.12	*PM2.5 < 56.43*	YES	YES	0.96
humidity > 79.15	*light < 4490.08*	YES	NO	0.95
light=high	*PM2.5 < 56.43*	YES	YES	0.95
humidity > 79.15 & *PM2.5 < 56.43*	*light < 4490.08*	YES	NO	0.93

Table 5: *Summary of association rules found by Apriori and FP-Growth*

4 Data Collection

This chapter describes how data have been collected in order to produce a dataset and test algorithms on it. The decision was to emulate the structure of the first tested dataset as much as possible, and this was possible by gathering informations of the local environment using both sensors and online weather API provided by **OpenWeather**.
A **Node-RED** application was deployed on a Raspberry Pi machine and it was used to retrieve data from the different sources, merged together and sent as a unique message to a Spark process by using Kafka. The Raspberry Pi was also connected with a USB cable to an Arduino Uno, which was in charge of controlling the various sensors. Two different versions of the Node-RED flow have been deployed, one named 'fast' which collects data every 15 seconds and the other named 'slow' which, instead, gathers data every 15 minutes. These two version can be run simultaneously and they produce two different datasets but, as regards all the results written in this report, in the end only the 'slow' one has been tested and used.

4.1 Node-RED

The Node-RED application is composed by one main flow which is in turn divided into 3 subflows:

- *'Get Weather Data'*;
- *'Get Sensor Data'*;
- *'Send to Kafka and slow dataset'*.

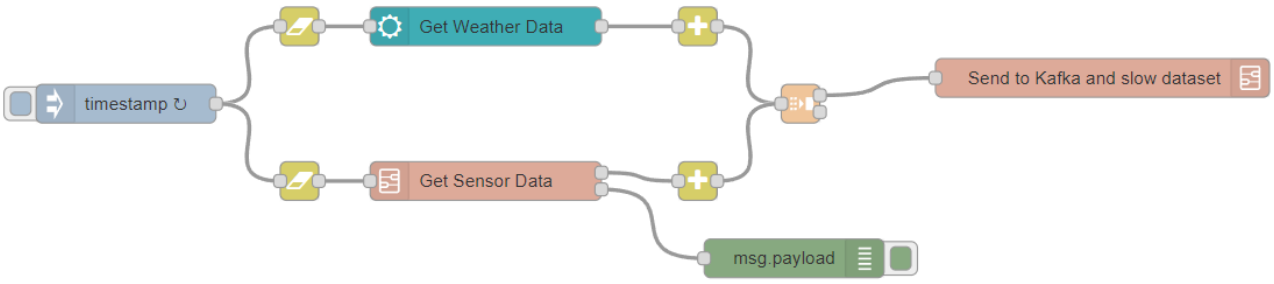


Figure 17: *The main flow*

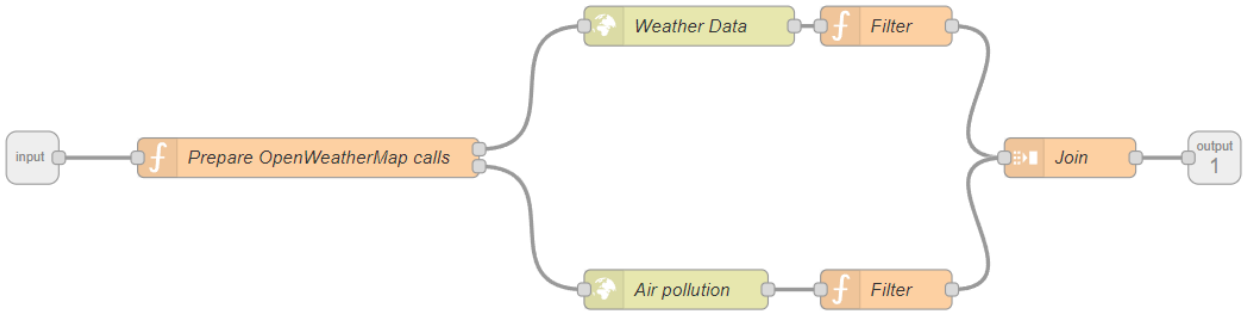
The first node is an *Inject* in charge of giving the beat to the application: it sends a message with the current timestamp every 15 minutes. The frequency was chosen in order to be neither too slow (so to produce a good amount of dataset rows in the slightest period possible) or too fast, as there's no point in measuring data so often because they change slowly, moreover OpenWeatherMap offers free API calls only behind a certain daily frequency.
The tick of the inject node enables two parallel threads: one asks for data to weather API (Par. 4.1.1) and the other sends commands to the Arduino board (Par. 4.1.2) to get sensor data. Messages coming out asynchronously from these two subflow are then merged in a *'Join'* function node (the salmon-colored square-shaped one on the right in Fig. 17). This function node acts like a filter, since it allows messages to pass only if both types of data have been received. The collected data are discarded in case the Arduino board is not attached or if there is no response from Weather API.
If both data are correctly received, they're passed to the final subflow, discussed in Par. 4.1.3, which is in charge of storing the whole payload inside a new row of the dataset and/or forwarding it to the Spark Session via Kafka.

4.1.1 Weather Data Acquisition

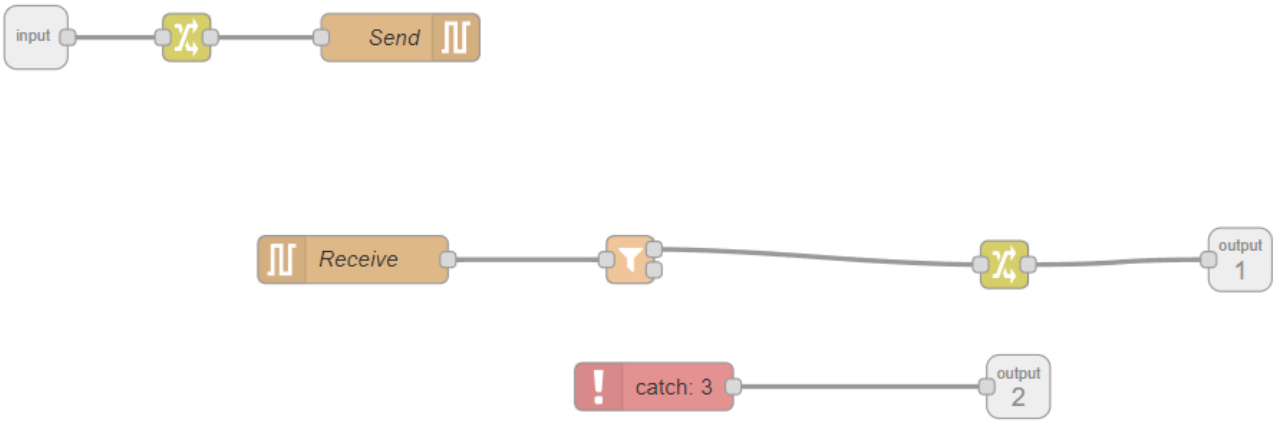
This subflow is in charge of acquiring weather data from online API (the turquoise one in Fig. 17) was necessary since the Arduino boards and the sensors attached to it were not able to capture some of the required data types to form the dataset. Its implementation is showed in Fig. 18a and it starts with a function that sets our local geographic coordinates and use them to prepare two different asynchronous API calls:

- the first returns a plethora of weather-related info, like humidity, temperature (real and perceived), wind informations, atmospheric conditions, eventual alerts, presence of clouds, rain, snow or fog, time of sunset and sunrise and the forecast for the next 24 hours;

- the other returns specific data about air quality, such as concentration of particulates and particular gases like ozone, carbon dioxide, ammonia etc.



(a) OpenWeatherMap API flow



(b) Subflow interfacing with the Arduino board

Figure 18: *The two parallel subflows gathering data from different sources.*

As regards weather data, they have to be filtered and mapped in a smaller subset of informations. A *'Filter'* function node deals with these tasks and it also calculates the **Beaufort** value of wind force from the received wind speed. Likewise, a homonym function placed downstream the air pollution call is in charge of selecting and forwarding only the data related to particulate concentration.
These data are finally merged together by the *'Join'* node and returned to the main flow.

4.1.2 Sensor Data Acquisition

This subflow deals with the activation command for the Arduino board to sense data from the local environment and sends it back. Communication between Node-RED and Arduino occurs through a serial channel. At first, an *'s'* or *'f'* character (depending on the type of flow) is sent to the Arduino board , that receives it and interprets it as a sense command request. The Arduino board answers with a *JSON*-formatted string, which is parsed inside a JSON object and then forwarded to the main flow. The implementation of this subflow is showed in Fig. 18b, while the details about how the Arduino board receives the message, activates sensors and sends the data back to Node-RED are explained in Par. 4.2.

4.1.3 Dataset population and forwarding to Spark

The last part of the Node-RED flow starts after that all the info coming from different sources have been collected and merged together to form a unique JSON object. This JSON is passed to the subflow named *'Send to Kafka and slow dataset'* (see Fig. 17). This subflow is actually splitted in two different parts, as it can be seen in Fig. 19:

- the upper part creates (if it doesn't exists) and populates the dataset with new rows for each incoming JSON object, then the new file is synchronized with its counterpart inside a remote cloud folder;

- the lower part is used to simulate sensor misreadings or other failures that compromise the integrity of the JSON object, then the result is sent through **Kafka** to a Spark session running on another machine.

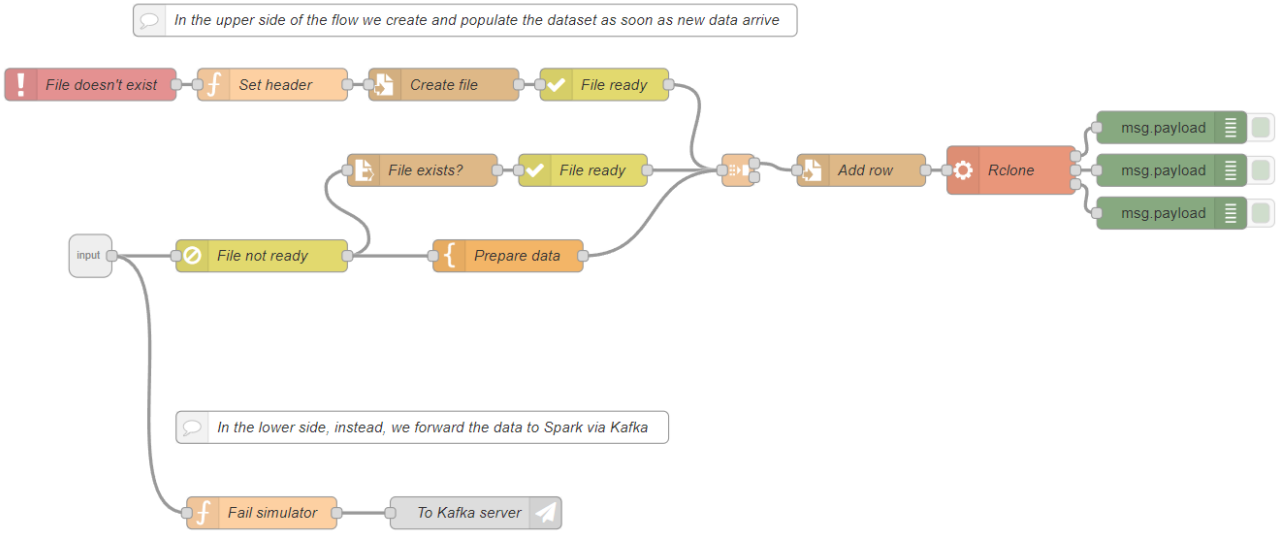


Figure 19: *This is the last subflow: once the JSON object arrives here, it's forwarded to Kafka and stored in the dataset.*

At first, when the subflow is activated, a flag named *'fileReady'* is set to false. This flag is needed to synchronize two parallel tasks that are put downstream. The first one is composed of just one template node (the orange one) that deals with setting the right order of informations arrived before storing them in a new row of the dataset (i.e., timestamp has to be the first value, then it comes humidity, light etc.), then the reordered object is forwarded to the square-shaped function node, which acts as a *Join*.
The other task starts with a control of the existence of the dataset file, performing different operations based on the outcome of the check:

- if it exists, then we can simply set our *fileReady* flag to true and forward this information to the downstream join function.
- if it doesn't exist, an exception is thrown and caught by the red node on the top left corner in Fig. 19. The file needs to be created before storing the row and this is done by setting, as first row of the new .csv file, the header containing names of the different columns. At this point we know for sure that the file exists, so we can put our flag to true and proceed with the execution of the flow.

This check is needed in order to not lose data in case the dataset doesn't exist yet. The square-shaped join function node receives the reordered data from the first parallel task and stores them, waiting to receive an answer from the other task; once this message is received and the flag is confirmed to be true, then the row can be appended at the bottom of the .csv file.
Finally the flow triggers an **RClone** script. RClone is a command-line program used to manage files on a cloud storage. In this case, this script is linked with a file inside a **Google Drive** folder and is in charge of constantly replacing this file with our freshly updated .csv dataset file.
As regards the lower part of the subflow, the function node named *'Fail simulator'* simply sets some random value to null in order to fake a wrong data reading: in this way Spark will be put in condition to predict the missing value. In the end, the JSON object is sent as a string to a Kafka server put on another machine.

4.2 Sensor data retrieval with Arduino

The physical architecture of the system is shown in Fig. 20 and is composed, as previously said, by a Raspberry Pi board linked to an Arduino UNO with a USB cable. The Arduino also communicates with two attached sensors, using the **I²C** protocol for the **BME280** and an analog connection with the light sensor.
The sketch for the Arduino board was programmed using **PlatformIO** extension for Visual Studio Code. It's a very simple sketch, consisting of two main methods:

- *setup()*, which is executed at the start and is in charge of configuring the ports for the two attached sensors and the speed of the Serial communication;

- *loop()*, that is continuously listening for new commands on the Serial port. It can detects two different commands, *'s'* and *'f'*, which stand for *slow* and *fast* and refer to the two different modes the Node-RED flow can be run.

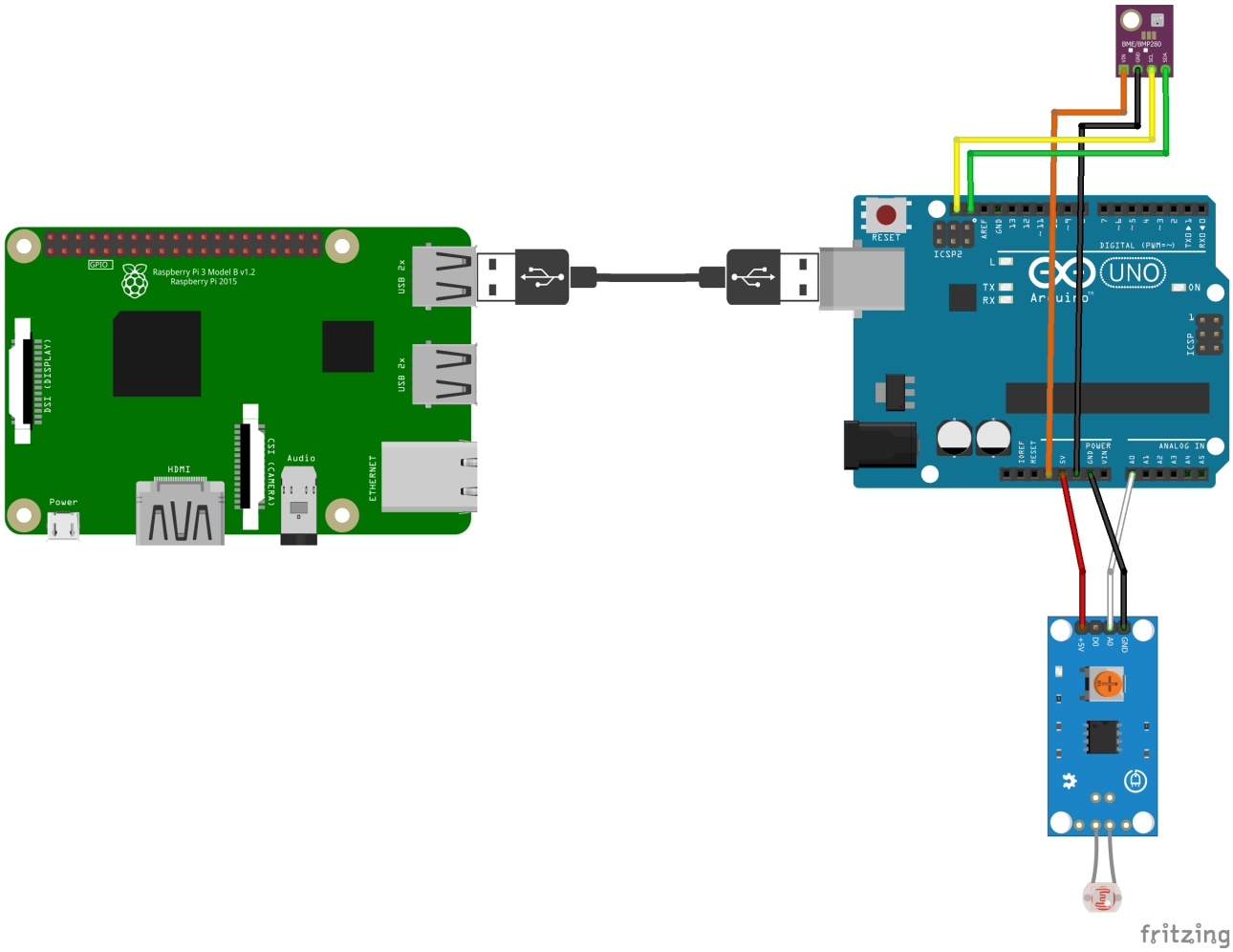


Figure 20: *Physical architecture of the system, with boards and sensors*

After receiving the command, the sensor data collection process starts, retrieving temperature, humidity and atmospheric pressure from the BME280 sensor and light intensity from the light sensor. These data are finally grouped together in a json object using *ArduinoJSON* library, together with an element named 'call' that specifies from which Node-RED flow the initial command was invoked (its value can be 0 if the command came from the 'slow' flow or 1 otherwise). The produced json object is finally serialized and delivered through the serial port to the Node-RED flow, then the board is ready again to receive a new message.

5 Analysis of generated dataset with Weka

After collecting data for a period and populating the dataset with a sufficient amount of rows, it has been analyzed using **Weka**. In particular, our effort consisted in exploring different algoritms, trying to identify eventual clusters, classifying temperature and humidity based on the other data and mining existing association rules between elements of the dataset.

5.1 Dataset structure and first preprocessing

The original dataset, as created and populated by sensor data retrieval process, has 3731 rows of data collected every 15 minutes (from 6th of March to 27th of April) and is made of the following columns:

- *timestamp*, in **UNIX** format. It specifies the exact moment in which a specific data was collected;
- *humidity*, a percentage number measuring the quality of air, from 0 (very dry) to 100 (very wet);
- *light*, a measure of intensity of light, from 0 (dark) to 100;
- *pm10*, it's a measure expressing the quantity, expressed in $\mu g/m^3$ (micrograms per cubic meter), of fine dust and particulate present in the air with a diameter not greater than 10 μm;
- *pm2.5*, same as before, but related to particulate with diameter not greater than 2.5 μm;
- air *pressure*, expressed in kPa ;
- *rain* volume for the last hour since the measure, expressed in mm;
- air *temperature* in Celsius degrees;
- *wind_dir* is a value that goes from 0 to 7 and specifies the aproximate direction of the wind:
  0. North;
  1. North-West;
  2. West;
  3. South-West;
  4. South;
  5. South-East
  6. East;
  7. North-East;
- *wind_dir_degrees*, another way of measuring wind direction, but expressed in sexagesimal degrees;
- *wind_force* expresses wind speed using the **Beaufort scale**;
- *wind_speed*, same as before but using m/s;

Once data have been collected, the produced dataset was analyzed in its parts in order to understand what's useful and what is not. At first, *timestamp* was recognized to be a source of potential informations about the portion of the day in which data was collected, but it was not exploitable by knowledge discovery algorithms in its current form, since timestamp, as-is, is a too much discriminating data (every row has a different value, so it behaves like an id). In order to remove its discriminating power while keeping useful nested informations, a Python script was developed and used with these key aspects:

1. it has to read the original dataset and produce a new one with the same number of rows;
2. *timestamp* has to be removed and replaced with a new column, named *timerange*, containing the range of hour in which the data was collected;
3. all the other values are kept identical.

As regards the choice made for these tests, days have been splitted in 12 2-hours long ranges. Other infos extractable from the timestamp (for example day of week, or month, or season, or minutes) have been discarded as considered useless, since measurements have been collected for a span of less than 2 months.
The new dataset, after being produced by the Python script and before being used by any of the algorithms that will be described in the next paragraphs, was analyzed with the Weka tool and pre-pruned of all not very useful infos, like:

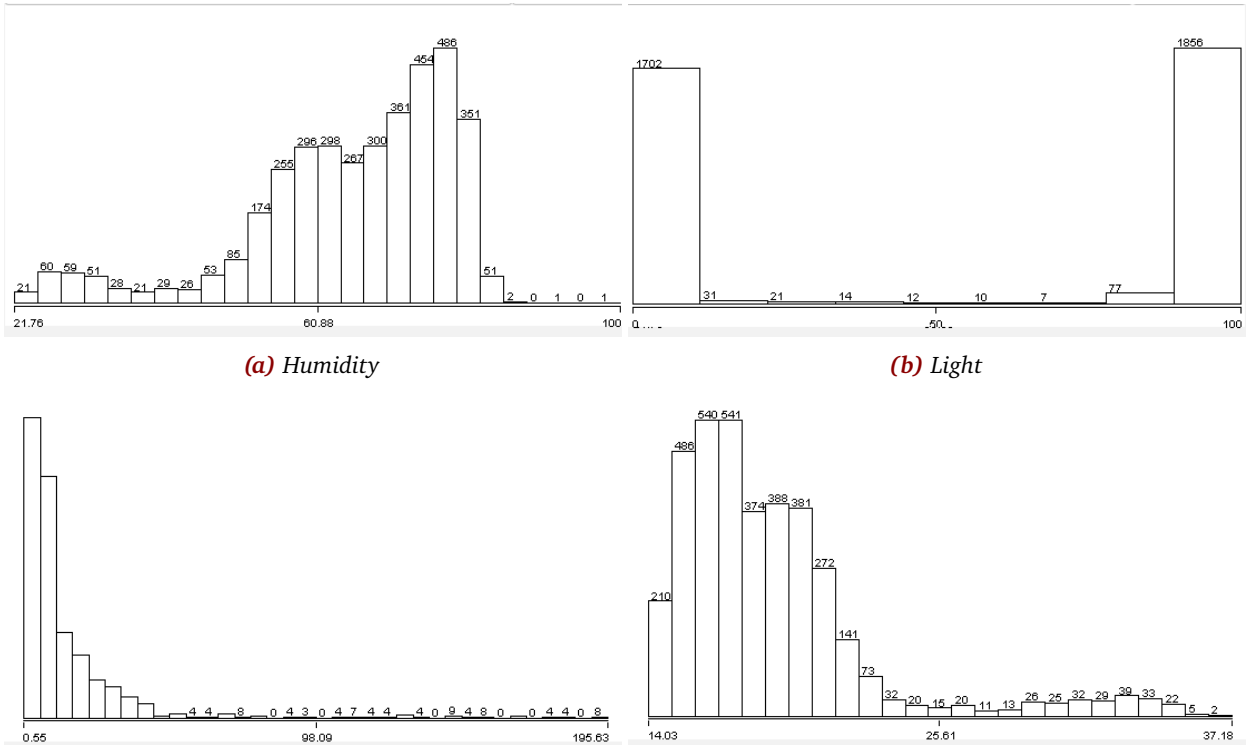
- *pressure*, because it turned out to be a pretty constant value, with measurements between 94 and 96 kPa, so it was too much characterizing, on the contrary of timestamp;
- *pm10*, because it was a sort of duplicate as its trend was practically identical to pm2.5;
- *rain*, as it almost never rained during the period in which data has been collected;
- *wind_dir_degrees*, because it's a duplicate of *wind_dir* but the latter is better sampled;
- *wind_speed*, for the same reasons described in the previous point, preferring to keep *wind_force*.

5.2 Application of clustering algorithms

After the preprocessing phase, the dataset was ready for the application of clustering algorithms using Weka. The dataset was initially made of the following elements:

- *time_range*, a Nominal element with 12 labels, where each label represent a span of 2 hours over one day;
- *humidity*, a Numeric element with values from 21.76 to 100;
- *light*, Numeric, with a diametrical distribution;
- *pm2.5*, Numeric, with a very high concentration of rows with low values;
- *temperature*, Numeric, with values from 14 °C to 37 °C;
- *wind_force*, Numeric.

After noticing poor performance results on the application of the algorithm, *time_range* and *wind_force* have been removed too.



(a) *Humidity* ***(b)*** *Light*

(c) *PM2.5* ***(d)*** *Temperature*

Figure 21: *Distribution of the dataset for the different elements*

5.2.1 K-means

The **K-means** is a clustering method that aims to partition elements in k groups, named *clusters*. The result of the algorithm depends on the choice of k and of the initial k centroids. Centroids are used to determine the belonging of an element to a specific cluster. This algorithm is natively supported by Weka, allowing the user to specify the initial k value and other parameters (like, for example, the distance function to be used), but it automatically puts centroids in randomic positions, without letting the user customize this aspect.
Before executing the algorithm, the dataset has been splitted in two parts:

- $\frac{2}{3}$ of it will be used as training set;
- the remaining $\frac{1}{3}$ is used as test set.

The algorithm has been tested several times, each time varying the **K** value, with a final collection of results that goes from k=1 to k=14. Putting k=1 is useless, as it creates a single cluster, but it was helpful to better analyze the trend of the *Internal Measure Curve* (Fig. 22). The Table 6 summarizes the results of the various executions of the algorithm and it was the starting point for the creation of the Internal Measure Curve.

Clusters (K)	Training Set			Test Set		
	SSE	Decreasement	Iterations	SSE	Decreasement	Iterations
1	1119,36	-	1	732,46	-	1
2	245.28	874,08	4	157.03	575,43	5
3	140.08	105,2	13	89.66	67,37	12
4	110.06	30,02	12	73.39	16,27	11
5	87,17	22,89	31	69,26	4,13	9
6	81,70	5,47	40	50.31	18,95	6
7	78,56	3,14	39	46,87	3,44	28
8	59,84	18,72	38	44,80	2,07	28
9	54,92	4,92	33	33.34	11,46	28
10	37,47	17,45	33	24,19	9,15	28
11	35,59	1,88	33	22.11	2,08	28
12	34,48	1,11	25	21,44	0,67	35
13	33,51	0,97	19	19,36	2,04	35
14	30,39	3,12	19	18,68	0,68	28

Table 6: *Comparison or K-Means results with different settings for the parameter K.*

At every test performed, **SSE** and number of iterations that the algorithm needed before its completion were noted and successfully taken into consideration to establish which **K** was giving the best results. Another useful parameter that can be obtained trivially is the decreasement of **SSE** from the previous test, which can be calculated in this way:

$$Decreasement(K) = SSE(K-1) - SSE(K)$$

The **SSE (Sum of Squared Errors)** is a statistical index useful for comparing two clusterings or two clusters: the lower it is, the better is the clustering. It can be also used to determine which is the optimal choice for **K** by directly looking at its trend on Fig. 22.

The **Internal Measure Curve** graph can indeed easily be exploited to determine the optimal value of K thanks to a rule known as *Elbow method* (or *Knee method*). This type of graph has a particular structure, as it always starts with a strong slope followed by a sudden flattening that makes the curve goes almost horizontally from that point on. The point where the inclination of the curve rapidly changes is named *elbow* or *knee* and it matches with the optimal value of **K**. In our case, the optimal value stands between 2 and 3, so our choice was for value 3. Choosing higher values is not useful because there is no more a significant improvement in **SSE** index. The graph also shows how the test set is representative of the training set, since the two curves follow the same trend.

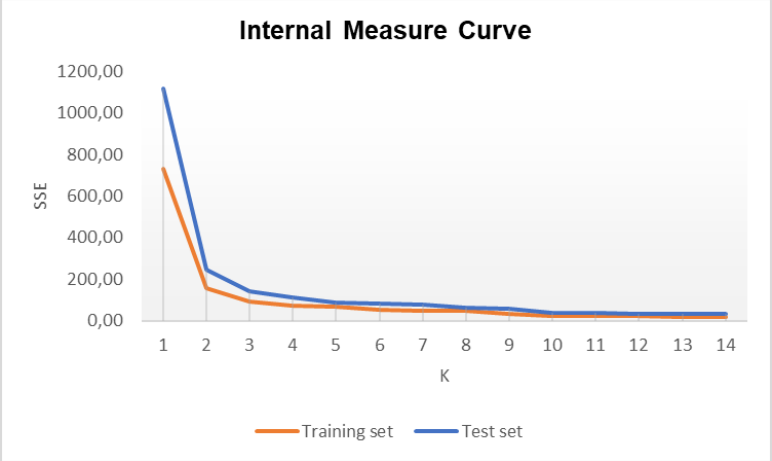
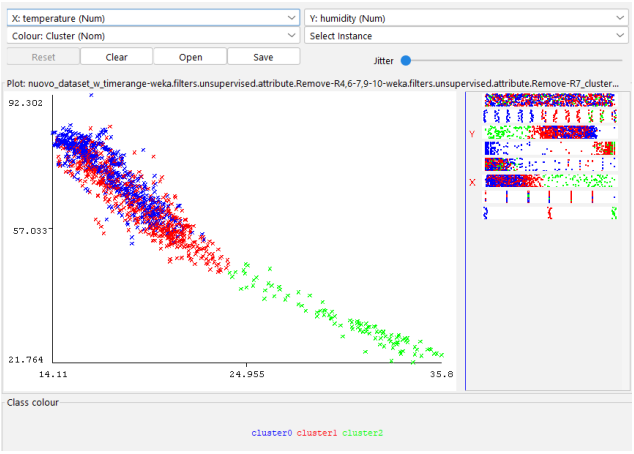


Figure 22: *Internal Measure Curve*

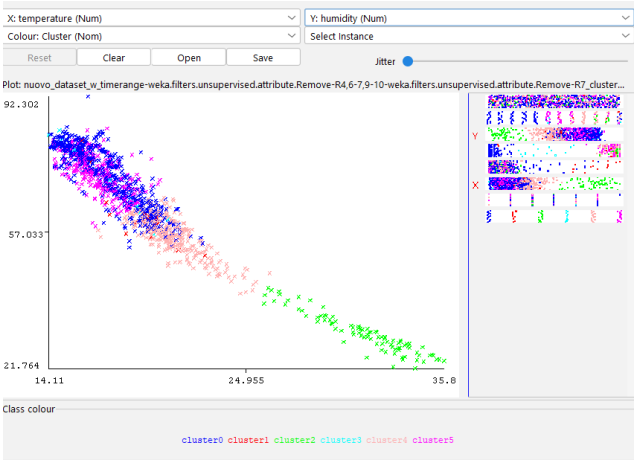
In Fig. 23a is shown the distribution of elements over the three clusters. While cluster 2 (the green one) is clearly distinguishable and separated from the others, the red and blue ones seem to be grasped together, so we

decided to double up the number of clusters in order to have better separated areas to be merged back in a second moment. The result is shown in Fig. 23b, but it didn't help, because only the green cluster is clearly separated from the others. In order to clearly define the other clusters and understand their boundaries, we need to also analyze other dimensions. In Fig. 23c temperature and pm2.5 were compared. Here we can also understand the boundaries of the red cluster, whose elements are present only for high values of pm2.5 and low temperatures. The remaining four clusters (blue, purple, azure and pink) are finally clearly visible by looking at Fig. 23d.
So we can establish these boundaries for the 6 clusters:

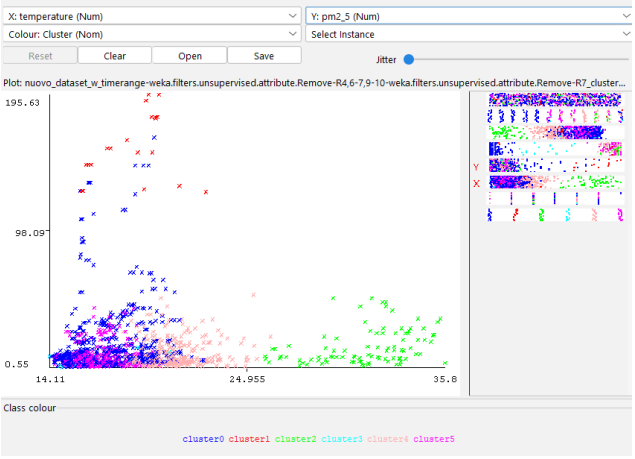
1. Green cluster contains all the elements with temperature $> 25\ °C$ and humidity < 40
2. Red cluster contains elements with pm2.5 $> 120\ \mu g/m^3$;
3. Blue cluster contains elements with pm2.5 $< 120\ \mu g/m^3$, temperature $< 22\ °C$ and light < 25;
4. the Purple one has elements with temperature $<20\ °C$ and light > 75;
5. the pink one is also found when light is more than 75, but at a temperature between $20°C$ and $26°C$;
6. finally, the azure one has values scattered at low temperatures and light values between 25 and 75.



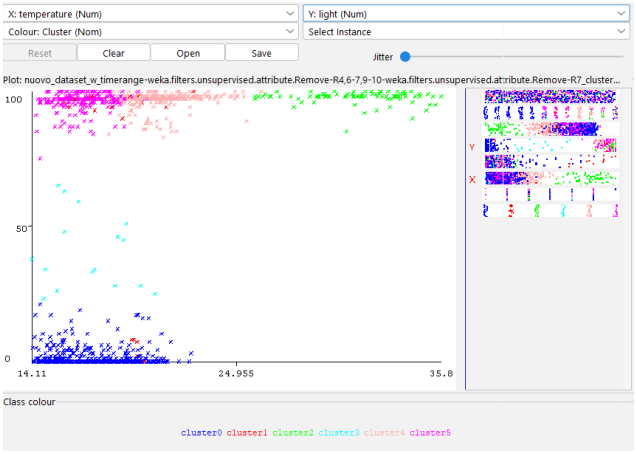
(a) Distribution over 3 clusters, taking into consideration temperature and humidity



(b) Distribution over 6 clusters, taking into consideration temperature and humidity



(c) Distribution over 6 clusters, taking into consideration temperature and pm2.5



(d) Distribution over 6 clusters, taking into consideration temperature and light

Figure 23: *Distribution of elements in clusters as result of K-means*

5.3 Application of classification algorithms

The columns used to test classification algorithms are: temperature, humidity, light, wind_force, pm2.5 and timerange. Before applying algorithms, a further preprocessing step was needed, in order to perform a *discretization* of column values. Discretization is an operation that reduces variety of values of a dataset, transforming numeric

columns in nominal ones. It's like categorizing all the elements into a specific subset of groups. The resulting columns have been structured in the following way:

- *humidity*, discretized in 4 bins (or rather possible categories);
- *temperature*, also discretized in 4 bins;
- *light*, divided in 2 bins, as there are obviously two highly-separated big sections as regards the distribution of this data, as it can be seen in Fig. 21c;
- *pm2.5*, divided in 3 bins;
- *wind_force*, that is already a discretized column (6 bins), since **Beaufort** values are integers;
- *timerange*, also already discretized in 12 bins.

Weka offers many more solutions and classification algorithms, so experiments with them have been more than the ones performed for clustering. we've been able to test 4 different algorithms:

1. Naive Bayes;
2. K-Nearest Neighbour;
3. Random Forest;
4. K-Star;

For each algorithm, multiple executions with different parameters have been tested.

```
=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        3291               88.2306 %
Incorrectly Classified Instances       439               11.7694 %
Kappa statistic                          0.7431
Mean absolute error                      0.077
Root mean squared error                  0.2154
Relative absolute error                 34.5187 %
Root relative squared error             64.5515 %
Total Number of Instances             3730

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0,918    0,107    0,954      0,918    0,936      0,791    0,963     0,985     '(-inf-19.8175]'
                 0,856    0,080    0,751      0,856    0,800      0,742    0,948     0,787     '(19.8175-25.605]'
                 0,210    0,002    0,765      0,210    0,329      0,391    0,976     0,532     '(25.605-31.3925]'
                 0,957    0,023    0,619      0,957    0,751      0,759    0,987     0,593     '(31.3925-inf)'
Weighted Avg.    0,882    0,095    0,891      0,882    0,879      0,765    0,961     0,911

=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 2430  217    0    0 |    a = '(-inf-19.8175]'
  116  702    2    0 |    b = '(19.8175-25.605]'
    0   16   26   82 |    c = '(25.605-31.3925]'
    0    0    6  133 |    d = '(31.3925-inf)'
```

(a) *Temperature*

```
=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        3136               84.0751 %
Incorrectly Classified Instances       594               15.9249 %
Kappa statistic                          0.6857
Mean absolute error                      0.1073
Root mean squared error                  0.2454
Relative absolute error                 40.7128 %
Root relative squared error             67.6358 %
Total Number of Instances             3730

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0,972    0,004    0,942      0,972    0,957      0,954    0,998     0,979     '(-inf-41.322752]'
                 0,824    0,087    0,753      0,824    0,787      0,716    0,945     0,831     '(41.322752-60.881835]'
                 0,902    0,247    0,865      0,902    0,883      0,666    0,897     0,920     '(60.881835-80.440917]'
                 0,000    0,000    ?          0,000    ?          ?        0,856     0,245     '(80.440917-inf)'
Weighted Avg.    0,841    0,179    ?          0,841    ?          ?        0,913     0,867

=== Confusion Matrix ===

    a    b    c    d   <-- classified as
  243    7    0    0 |    a = '(-inf-41.322752]'
   15  748  145    0 |    b = '(41.322752-60.881835]'
    0  234 2145    0 |    c = '(60.881835-80.440917]'
    0    4  189    0 |    d = '(80.440917-inf)'
```

(b) *Humidity*

Figure 24: *Result details for Naive Bayes*

5.3.1 Naive Bayes

This is the simplest algorithm that can be performed using Weka, since there are no parameters to set. It has been tested in order to evaluate the correct classification of values of temperature and humidity, with result details showed respectively in Fig. 24a and 24b.
We can notice an important results as regards the number of correctly classified instances for both temperature (88.23%) and humidity (84.07%), but there are problems in both cases:

- the algorithm has difficulties in classifying values of temperature between 25 °*C* and 31 °*C*, corresponding to the third bin, since only 21% of them was correctly classified, as stated by the relative **TP Rate**;
- it also has difficulties in classyfing values of humidity higher than 80%, with a terrible result of 0% of correctly classified instances for the fourth bin of humidity.

We tried to improve the classification in this particular cases altering the matrix of costs for both cases, favouring and incentivating the algorithm to behave correctly when meeting those values. **Matrix of costs** is a n*n matrix of coefficients that are applied to the algorithm in order to alter its natural behaviour, as the algorithm will always choose the result with the least resulting cost. The total cost is calculated performing the sum of each element of the **Confusion matrix** $\mathcal{M}$ (present in the bottom-left corner of Fig. 24a and 24b) multiplied by the element in the same position in the Matrix of costs $\mathcal{C}$.

$$TotalCost = \sum_{i=1}^{n}\sum_{j=1}^{n}(\mathcal{M}_{ij} \cdot \mathcal{C}_{ij})$$

After various attempts, these are the configurations for $\mathcal{C}$ that gave best results:

$$\mathcal{C}_{temp} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1.5 & 0 & 4.4 \\ 1 & 1 & 2.3 & 0 \end{bmatrix} \qquad \mathcal{C}_{hum} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 8 & 0 \end{bmatrix}$$

The application was successful in case of humidity classification, with the correctly classified instances of the 4th bin raised up to 60%, at the cost of reducing the total overall score from 84% to 78.65%. As regards temperature, instead, we were not able to obtain acceptable results for the 3rd bin, as we managed to increase them up to 49.2%, but with a consequent reduction of the 4th bin to 51.8% and an overall score of 87.53%. Forcing the cost matrix even more was causing the 4th bin to drop below the threshold of 50%, so it wasn't useful.

5.3.2 K-Nearest Neighbour

Weka offers a high level of customization for this algorithm, starting from the choice for value **K** to the selection of different distance functions and search algorithms. At first, we tried to understand what was the best values for K. It resulted to be 7 for humidity and 3 for temperature, although the highest scores were obtained with K=1 for both cases. Table 7 shows all the details of the various results.

Type	K	Search Algorithm	TP Rate				Correct Classifications
			Bin 1	Bin 2	Bin 3	Bin 4	
Temperature	1	LinearNN	0.940	0.793	0.315	0.899	88.52 %
	3	LinearNN	0.940	0.795	0.282	0.899	88.44 %
Humidity	1	LinearNN	0.988	0.801	0.937	0	85.84 %
	7	LinearNN	0.980	0.784	0.942	0	85.76 %
	7	KDTree	0.988	0.791	0.943	0	86.01 %

Table 7: *Comparison or KNN results with different settings*

The best result, as regards, temperature, was obtained by changing the distance function of LinearNN search algorithm. All the various attempts are summarized in Table 8. Although best results have been obtained with

K=1, LinearNN search algorithm and Filtered Distance function, we decided to keep K=3 because a higher value for K makes the overall algorithm less vulnerable to outliers. Same concept has been applied to humidity, where the preferred choice has been K=7.

K	Distance Function	Correct Classifications
1	Euclidean	88.52 %
	Chebyshev	88.55 %
	Filtered	88.61 %
	Manhattan	88.52 %
	Minkowski	88.52 %
3	Euclidean	88.44 %
	Chebyshev	88.31 %
	Filtered	88.39 %
	Manhattan	88.44 %
	Minkowski	88.44 %

Table 8: *Comparison or different distance functions for linearNN applied to Temperature classification*

Again, like in the previous algorithm, the KNN suffers to correctly classify the 3rd bin of temperature and the 4th bin of humidity. Again, in order obtain more balanced results, we tried to apply the following cost matrices:

$$\mathcal{C}_{temp} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 3.1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \qquad \mathcal{C}_{hum} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 10 & 0 \end{bmatrix}$$

Below is a recap of the final results after altering the cost matrix.

Type	K	Search Alg.	Distance F.	TP Rate				Correct Classifications
				Bin 1	Bin 2	Bin 3	Bin 4	
Temperature	3	LinearNN	Euclidean	0.940	0.795	0.540	0.525	87.91 %
Humidity	7	KDTree	-	0.988	0.794	0.831	0.518	81.63 %

Table 9: *Final KNN results after altering the cost matrix*

5.3.3 Random Forest

Random Forest, like Bayes, is an algorithm that cannot be parameterized in Weka but, despite that, it produced very good results, which can be seen in Table 10.

Type	Cost Matrix	TP Rate				Correct Classifications
		Bin 1	Bin 2	Bin 3	Bin 4	
Temperature	Plain	0.939	0.799	0.331	0.899	88.66 %
	Altered	0.939	0.799	0.516	0.547	87.96 %
Humidity	Plain	0.992	0.785	0.940	0	85.71 %
	Altered	0.992	0.791	0.839	0.508	82.06 %

Table 10: *Comparison or KNN results with different settings*

We were able to obtain balanced results for both temperature and humidity using these cost matrices:

$$\mathcal{C}_{temp} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 3 \\ 1 & 1 & 1 & 0 \end{bmatrix} \qquad \mathcal{C}_{hum} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 10 & 0 \end{bmatrix}$$

5.3.4 K-Star

As already mentioned in Par. 3.3.4, is important to choose a good value for the *blend* parameter of this algorithm, in a similar way as already done with the choice of K in KNN algorithm. After various attempts, the best results for temperature and humidity have been obtained with a blend value of respectively 17 (Fig. 25a) and 16 (Fig. 25b).

```
=== Classifier model (full training set) ===

KStar Beta Verion (0.1b).
Copyright (c) 1995-97 by Len Trigg (trigg@cs.waikato.ac.nz).
Java port to Weka by Abdelaziz Mahoui (aml4@cs.waikato.ac.nz).

KStar options : -B 17 -M a

Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        3325               89.1421 %
Incorrectly Classified Instances       405               10.8579 %
Kappa statistic                          0.753
Mean absolute error                      0.0932
Root mean squared error                  0.2024
Relative absolute error                 41.8002 %
Root relative squared error             60.6321 %
Total Number of Instances             3730

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure  MCC      ROC Area  PRC Area  Class
               0,950     0,159     0,936      0,950     0,943      0,800     0,969     0,987     '(-inf-19.8175]'
               0,790     0,052     0,812      0,790     0,801      0,746     0,958     0,861     '(19.8175-25.605]'
               0,306     0,004     0,717      0,306     0,429      0,458     0,975     0,483     '(25.605-31.3925]'
               0,899     0,019     0,648      0,899     0,753      0,753     0,987     0,566     '(31.3925-inf)'
Weighted Avg.  0,891     0,125     0,891      0,891     0,887      0,775     0,967     0,927

=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 2514  133    0    0 |    a = '(-inf-19.8175]'
  171  648    1    0 |    b = '(19.8175-25.605]'
    1   17   38   68 |    c = '(25.605-31.3925]'
    0    0   14  125 |    d = '(31.3925-inf)'
```

(a) *Temperature*

```
=== Classifier model (full training set) ===

KStar Beta Verion (0.1b).
Copyright (c) 1995-97 by Len Trigg (trigg@cs.waikato.ac.nz).
Java port to Weka by Abdelaziz Mahoui (aml4@cs.waikato.ac.nz).

KStar options : -B 16 -M a

Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        3203               85.8713 %
Incorrectly Classified Instances       527               14.1287 %
Kappa statistic                          0.7084
Mean absolute error                      0.126
Root mean squared error                  0.2358
Relative absolute error                 47.8308 %
Root relative squared error             64.9802 %
Total Number of Instances             3730

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure  MCC      ROC Area  PRC Area  Class
               0,984     0,004     0,943      0,984     0,963      0,960     0,998     0,978     '(-inf-41.322752]'
               0,744     0,036     0,868      0,744     0,801      0,747     0,955     0,874     '(41.322752-60.881835]'
               0,959     0,303     0,848      0,959     0,900      0,703     0,912     0,934     '(60.881835-80.440917]'
               0,000     0,000     ?          0,000     ?          ?         0,855     0,241     '(80.440917-inf)'
Weighted Avg.  0,859     0,202     ?          0,859     ?          ?         0,925     0,886

=== Confusion Matrix ===

    a    b    c    d   <-- classified as
  246    4    0    0 |    a = '(-inf-41.322752]'
   15  676  217    0 |    b = '(41.322752-60.881835]'
    0   98 2281    0 |    c = '(60.881835-80.440917]'
    0    1  192    0 |    d = '(80.440917-inf)'
```

(b) *Humidity*

Figure 25: *Result details for K-Star*

We notice that all the previous algorithms we tried had to deal with the same difficulties in correctly classifying some categories, and K-star didn't make any exception, also if it obtained the amazing result of 89.14% of correctly classified instances for temperature, the highest value ever obtained during the tests. Unfortunately, this time altering the cost matrix didn't solve the situation neither for humidity or for temperature. Best results (details on Table 11) were obtained with this configuration:

$$\mathcal{C}_{temp} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 2.5 \\ 1 & 1 & 1 & 0 \end{bmatrix} \qquad \mathcal{C}_{hum} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 10 & 0 \end{bmatrix}$$

Type	Blend	Cost Matrix	TP Rate				Correct Classifications
			Bin 1	Bin 2	Bin 3	Bin 4	
Temperature	17	Plain	0.950	0.790	0.306	0.899	89.14 %
		Altered	0.948	0.784	0.435	0.576	88.09 %
Humidity	16	Plain	0.984	0.744	0.959	0	85.87 %
		Altered	0.976	0.756	0.855	0.492	81.98 %

Table 11: *Comparison or K-Star results with different blend and cost matrix settings.*

5.3.5 Final recap

The best results of our experiments have been obtained with **Random Forest**, as it produced great balanced results (namely, results with each FP-Tree > 0.5). **K-Star** obtained the highest score, but it wasn't able then to convert it in a more balanced result, so it should be discarded in this case. A brief recap of obtained results is shown in Table 12.

Type	Algorithm	Best overall result	Best balanced result
Temperature	Naive Bayes	88.23 %	Not obtained
	KNN	88.61 % (K=1)	87.91 %
	Random Forest	88.65 %	87.96 %
	K-Star	89.14 %	Not obtained
Humidity	Naive Bayes	84.07 %	78.76 %
	KNN	86.01 % (K=7)	81.64%
	Random Forest	85.7%	82.06%
	K-Star	85.87 %	Not obtained

Table 12: *Recap of obtained results.*

5.4 Mining association rules

Experiments about mining association rules have been performed using both **Apriori** and **FP-Growth** algorithm. The *minimum support* has been set to 40 %, which, for a total number of rows equal to 3730, means that one association rule must be present in at least 1492 rows. *Minimum confidence*, instead, was set to 0.9. Table 15 recaps the results of both algorithms.

Association Rule		Algorithm		Confidence
Left Hand Side	Right Hand Side	Apriori	FP-Growth	
light > 50	*pm2.5 <65.57*	YES	YES	0.98
60.88 < humidity < 80.44	*pm2.5 <65.57*	YES	YES	0.96
temperature < 19.81 & 60.88 < humidity < 80.44	*pm2.5 <65.57*	YES	YES	0.96
temperature < 19.81	*pm2.5 <65.57*	YES	YES	0.96
wind_force = 2	*pm2.5 <65.57*	YES	YES	0.99
light < 50 & temperature < 19.81	*pm2.5 <65.57*	YES	NO	0.95
light < 50	*pm2.5 <65.57*	YES	NO	0.95
60.88 < humidity < 80.44	*temperature < 19.81*	YES	YES	0.95
60.88 < humidity < 80.44 & pm2.5 <65.57	*temperature < 19.81*	YES	YES	0.95
60.88 < humidity < 80.44	*pm2.5 <65.57 temperature < 19.81*	YES	YES	0.91

Table 13: *Summary of association rules found by the two algorithms*

In order to better understand the distribution and support of different itemsets, let's associate each 1-itemset to a letter, as shown below:

Letter	Itemset	Support
A	*light > 50*	1953
B	*60.88 < humidity < 80.44*	2379
C	*temperature < 19.81*	2647
D	*wind_force = 2*	1793
E	*light < 50*	1777
F	*pm2.5 <65.57*	3594

Table 14: *Support for 1-itemsets*

Each green letter in table states that the itemset is a **Closed** one. Red letters, instead, represent **Maximal Closed Itemsets**.

Itemset	Support
BF	2289
BC	2255
EF	1687
EC	1616
AF	1907
FC	2538
FD	1709
BFC	2168
EFC	1535

Table 15: *Support for 2-itemsets and 3-itemsets*

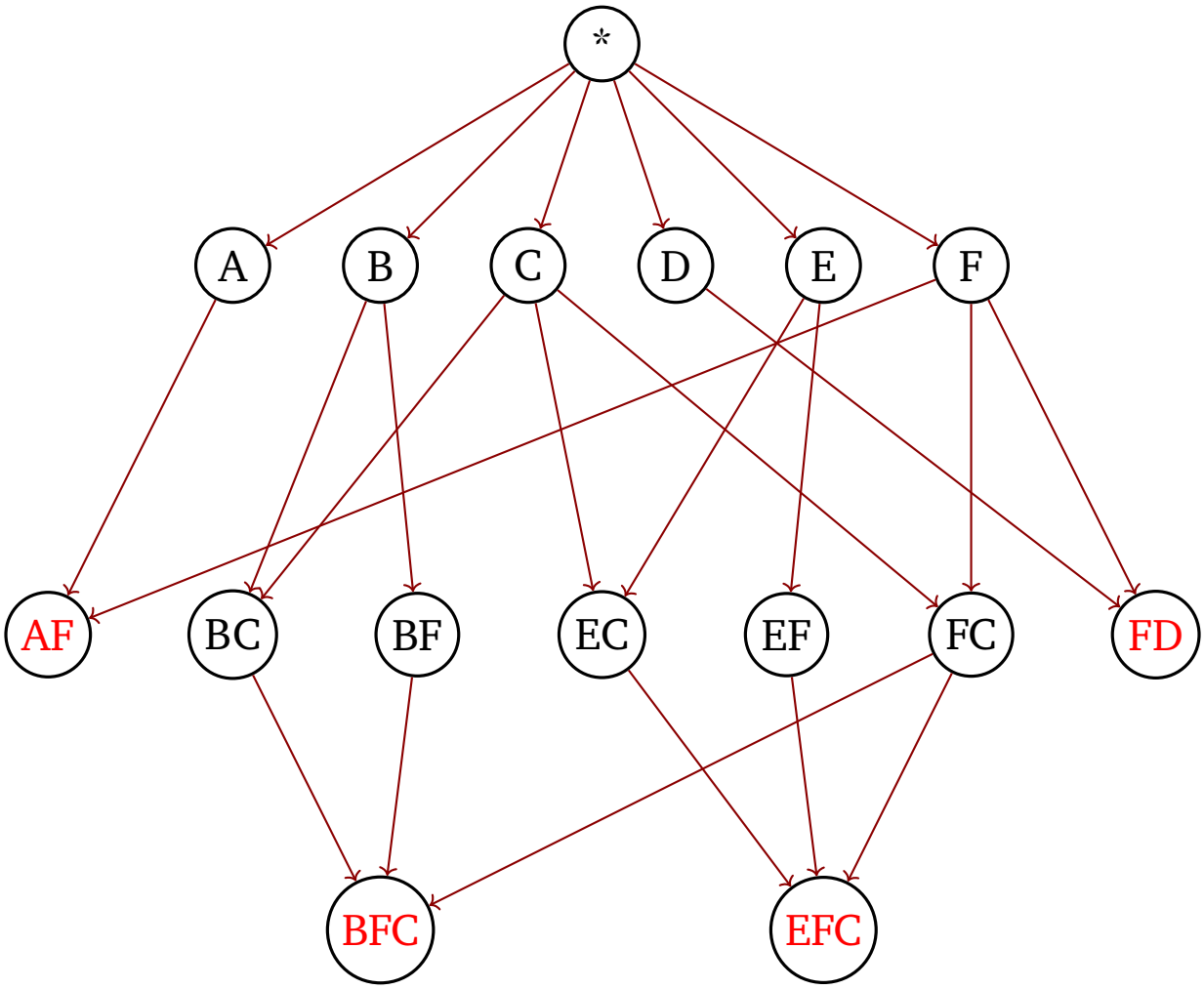


Figure 26: *Visual representation of the distribution of Closed Frequent Itemsets (where red ones are also Maximal).*

6 Implementation of Machine Learning Algorithms

As said before, the system is able to achieve a kind of fault tolerance level, in case a sensor fail or a value in the OpenWeather call is missing. When this happens, a machine learning algorithm is called, to predict the missing value from the reading. In the following paragraphs, the whole process flow will be shown.

6.1 PySpark

Given the fact that Apache Spark is written in Scala, in order to support the collaboration of Apache Spark and Python, was released PySpark, which is actually a Python API for Spark. PySpark allows the interface with Resilient Distributed Datasets (RDDs) in Apache Spark and Python programming language. This has been achieved by taking advantage of the Py4j library, which is integrated within PySpark and allows Python to dynamically interface with JVM objects (but, obviously, Java must always be installed on the machine to run the application). PySpark wraps up all the Apache Spark ecosystem so not only implements the Python counterpart of Spark Core, but also the counterparts of all the main Spark API's, such as Spark SQL, Spark MLlib and Spark Streaming. The use of Python instead of Scala, while may reduce some native capabilities given the fact that the application is not written in the same language as the framework, can have some benefit such as the bigger catchment area, that can help in the development of the application, and the more "natural" interoperability with other popular libraries such as NumPy, Scikit_Learn, Tensorflow ecc.

6.2 Project Structure

The Pyspark project is built in a virtual environment (venv), which contains all the necessary library for the application. The structure is composed by the data folder, which contains the dataset to be used in the regression algorithms, a TrainedModels folder, that will be contain all the model already trained and of which will be discussed about later in the report, a Main.py file that contains the KafkaConsumer entity and that will call, sequentially, all the other files that, as can be seen, contains each one a different kind of regression algorithm; moreover, there is also a CorrelationTest.py file, which contain a correlation test that show which is the correlation of the various features with regard of the current label upon which is performed the prediction.

6.3 Kafka Setup and Main method

In this project, the Apache Kafka platform is utilized to establish the communication between the Node-Red middleware and the machine learning algorithm. Apache Kafka is deployed and ran on the same machine (a Windows pc) that run the machine learning algorithm and, to execute all the following commands, it is necessary to be in the main folder of the Kafka version downloaded on the machine.

```
############################# Socket Server Settings #############################

# The address the socket server listens on. If not configured, the host name will be equal to the value of
# java.net.InetAddress.getCanonicalHostName(), with PLAINTEXT listener name, and port 9092.
#   FORMAT:
#     listeners = listener_name://host_name:port
#   EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://192.168.43.204:9094

# Listener name, hostname and port the broker will advertise to clients.
# If not set, it uses the value for "listeners".
advertised.listeners=PLAINTEXT://192.168.43.204:9094
```

Before starting to launch Kafka, it is necessary to perform some preliminary operations. In the config folder, the server.properties file must be modified, specifically uncommenting the listeners and advertised.listener variables and manually inserting the local IP address of the machine in which Kafka is launched, so that the broker and consequentially the topic, can be accessible from the outside. After doing so, and before starting with the real Kafka launching, it is also advised to delete every file contained in the log folder, which may cause some problem in the initialization of the server.
At this point it's possible to start with the setup of the Kafka platform, so first thing first, the zookeeper process is deployed, with the following command:

```
./bin/windows/zookeeper-server-start.bat config/zookeeper.properties
```

Then, in another terminal, the server process must be deployed, utilizing the following command:

```
./bin/windows/kafka-server-start.bat config/server.properties
```

At this point, it is possible to create the topic that will be used in this project:

```
./bin/windows/kafka-topics.bat --create --bootstrap-server 192.168.43.204:9094 --replication-factor 1 --
    partitions 1 --topic RaspTest
```

The Kafka Producer entity, as said in the previous chapter, is instantiated in the Node-Red middleware through a Kafka Producer node, that will write the message on the specified topic.
Now that everything has been set up, it is possible to launch the application, that as first operation, will instantiate the KafkaConsumer entity, that will keep listen to the specified topic until it will receive a message.
The consumer is initialized utilizing the kafka-python package that highly simplifies all the Kafka classic operations in a Python environment. The consumer takes as input the name of the topic, the IP address on which is deployed the bootstrap server, the auto_offset_reset property setted to latest to receive only the messages present in the topic from the moment in which the consumer in connected to the topic, onwards, and a deserializer to decode from byte the Json object contained in the message. Then, if the consumer is not null and there is a message incoming, this message is taken as input for the setLabel method, that will calculate which is the feature upon which perform the prediction, and will return it; otherwise, the application stops with a dedicated error message.

```
MarioPingitore *
def setupKafka():
    try:
        consumer = KafkaConsumer('RaspTest',
                                 group_id=None,
                                 auto_offset_reset='latest',
                                 bootstrap_servers=['192.168.43.204:9094'],
                                 value_deserializer=lambda m: json.loads(m.decode('utf-8')))
        if consumer is not None:
            for message in consumer:
                if message is not None:
                    incomingReading = message.value
                    print(incomingReading)
                    settedLabel = setLabel(incomingReading)
                    return settedLabel
    except Exception as e:
        print(traceback.format_exc())
        print(e)
        print("Kafka Connection Error -> Cannot establish connection to the broker's topic")
```

Figure 27: *The setupKafka method*

At this point, in the main, the setLabel (Fig. 28) method recognize the feature upon which operate the prediction by checking which is the feature with an unsuitable value and take that feature as a label. The latter one will be taken as input by the callRegressionTests method, which will call, sequentially, all the regression algorithms implemented in the project; moreover, before starting with the regression, a correlationTest is performed on the feature with respect to the label. At the end, when all the tests are finished, the application will restart itself automatically and will be again listening to the Kafka topic for incoming messages.

```
MarioPingitore
def setLabel(incomingReading):
    try:
        for key, value in incomingReading.items():
            if value == '' or value is None or value == "undefined":
                label = key
                return label
    except Exception as e:
        print(traceback.format_exc())
        print(e)
        print("Label Setting Error -> Cannot extract the label from incoming reading")


MarioPingitore *
def callRegressionTests(label):
    subprocess.call(['cmd', '/c', 'python LinearRegression.py', label])
    subprocess.call(['cmd', '/c', 'python DecisionTreeRegression.py', label])
    subprocess.call(['cmd', '/c', 'python IsotonicRegression.py', label])
    subprocess.call(['cmd', '/c', 'python RandomForestRegression.py', label])
    subprocess.call(['cmd', '/c', 'python GBTRegression.py', label])
```

Figure 28: *The setLabel method*

6.4 Regression algorithms

In the following paragraph, the regression algorithms will be showed. To not stretch unnecessarily, only the first algorithm will be explained with detail, since the major portion of the algorithms, that contains loading of the data, filtering, pre-processing and so on, is the same for each one, and so for the other files, only the different regression solution implemented will be highlighted.

6.4.1 Linear Regression

The linear regression algorithm is a type of supervised machine learning algorithm that predict the value of a dependent variable from and independent variable by fitting a linear line (in the form $y = mx + b$) in order to match as much as possible the relationship of dependent and independent variables based on a multitude of point provided to the model, similarly to a scatter plot; the slope of the line indicates how much the dependent variable changes for a unit, change in the independent variables. It is possible to measure how much the algorithm has performed well by looking at the RMSE, that is the parameter that will be utilized also in the other algorithms, to equally compare them. Now the algorithm will be explained in detail.
At first, the label is taken from the Main, and it is compared with a reference String containing all the features and the label is “subtracted” from the list, so that the latter will contains only the name of the features.

```
"""We vectorize the dataset utilizing the features and split the created dataframe  in train and test portions"""
MarioPingitore
def vectorizeAndSplit(dataDf, features):
    vectorAssembler = VectorAssembler(inputCols=features, outputCol='features')
    vDataDf = vectorAssembler.transform(dataDf)

    splits = vDataDf.randomSplit([0.7, 0.3])
    trainDf = splits[0]
    testDf = splits[1]
    return trainDf, testDf
```

Figure 29: *Vectorization method*

After this multiple operations are performed regarding the timestamp contained in the dataset. First, the dataset in converted from UNIX to a yyyy-MM-dd HH:mm:ss format; then, this new timestamp is filtered, by extracting only month, day and hour, that are the only relevant data useful for the regression and to remove all the redundancies of the other data; finally, the hour is aggregated in an hour range to achieve better performance results.

```
"""The Linear Regression model is prepared, as well as an evaluator and the GridSearch operation utilizing a ParamGrid
   both to be used in the CrossValidator operation. 5 folds is the optimal tradeoff between speed of training and
   accuracy of the results."""
 MarioPingitore
def linearRegression(label):
    try:
        lr = LinearRegression(featuresCol = 'features', labelCol=label)

        # Create ParamGrid for Cross Validation
        lrParamGrid = (ParamGridBuilder()
                       .addGrid(lr.regParam, [0.001, 0.01, 0.1, 0.5, 1.0, 2.0])
                       .addGrid(lr.elasticNetParam, [0.0, 0.25, 0.5, 0.75, 1.0])
                       .addGrid(lr.maxIter, [1, 5, 10, 20, 50])
                       .build())
        lrEvaluator = RegressionEvaluator(predictionCol="prediction", labelCol=label, metricName="rmse")

        # Create 5-fold CrossValidator with 3 level parallelism
        lrcv = CrossValidator(estimator = lr,
                              estimatorParamMaps = lrParamGrid,
                              evaluator = lrEvaluator,
                              numFolds = 5,
                              parallelism=3)
        return lrcv, lrEvaluator
    except Exception as e:
        print(traceback.format_exc())
        print(e)
        print("Model Creation Error -> Cannot perform the creation of the model")
```

Figure 30: *LinearRegression algorithm*

At this point, after having also cached the dataframe with the updated features, to speed up future accesses a vectorization (Fig. 29) is performed on the dataframe, that is also, sequentially, split in 2 random partitions in a 70/30 ratio, to be utilized respectively for the training phase and for the test phase. Finally, the LinearRegression (Fig. 30) entity is initialized: a ParamGrid is created to perform a GridSearch operation, to find the best combination of hyperparameters that will allow to achieve the best result possible. After creating also a RegressionEvaluator, the latter and the ParamGrid are taken as input by the CrossValidator, designed with 5 folds and 3 level of parallelism.

```
 MarioPingitore *
def trainOrLoad(label, lrcv, trainDf):
    modelPath = Path("./TrainedModels/LinearRegressionBestModels/"+label)
    if not modelPath.exists():
        try:
            lrModel = lrcv.fit(trainDf)
            bestModel = lrModel.bestModel
            print(bestModel)
            bestModel.write().overwrite().save("./TrainedModels/LinearRegressionBestModels/"+label)
            print("Model Saved")
        except Exception as e:
            print(traceback.format_exc())
            print(e)
            print("Model Saving Error -> Cannot perform the saving of the model")
        try:
            trainingSummary = bestModel.summary
            print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
            print("r2: %f" % trainingSummary.r2)
            return bestModel
        except Exception as e:
            print(traceback.format_exc())
            print(e)
            print("Training Operation Error -> Cannot perform the training of the model")
    else:
        try:
            bestModel = LinearRegressionModel.load("./TrainedModels/LinearRegressionBestModels/"+label)
            print("Model Loaded")
            return bestModel
```

At this point, the training operation can begin. The first time that a specific label performs the training, the model obtained through the fit method, is saved in a specific folder, so that the next time that a prediction must be performed on the same label with the same algorithm, the

system can load the already trained model and utilize the latter for the prediction phase, thus cutting greatly the operation time of the whole process.
Finally, both prediction and evaluation operations are performed utilizing the trained model and the test portion of the dataframe and, at the end, results about the RMSE are printed in output.

6.4.2 Isotonic Regression

The Isotonic (or Monotonic) regression is a regression technique in which the predictor variable is monotonically related to the target variable. This means that as the value of the predictor variable increases, the value of the target variable either increases or decreases in a consistent, non-oscillating manner. Mathematically, isotonic regression can be formulated as an optimization problem.

$$\sum_{i=1}^{n}(y_i - f(x_i))^2$$

in which the goal is to find a monotonic function that minimizes the sum of the squared errors between the predicted and observed values of the target variable. Regarding the implementation, it's pretty much the same as the Linear Regression, except for the ParamGrid utilized in the CrossValidator, in which the parameter isotonic is set to determine if the returning sequence should be increasing or decreasing, and in the Evaluation phase, in which additional metrics are showed.

```
def isotonicRegression(label):
    try:
        ir = IsotonicRegression(featuresCol='features', labelCol=label)

        # Create ParamGrid for Cross Validation
        irParamGrid = (ParamGridBuilder()
                       .addGrid(ir.isotonic, [True, False])
                       .build())
        irEvaluator = RegressionEvaluator(predictionCol="prediction", labelCol=label, metricName="rmse")

        # Create 5-fold CrossValidator with 3 level parallelism
        ircv = CrossValidator(estimator=ir,
                              estimatorParamMaps=irParamGrid,
                              evaluator=irEvaluator,
                              numFolds=5,
                              parallelism=3)
        return ircv, irEvaluator
    except Exception as e:
        print(traceback.format_exc())
        print(e)
        print("Model Creation Error -> Cannot perform the creation of the model")
```

6.4.3 Decision Tree Regression

Decision tree is a popular supervised machine learning algorithm that can be used for both regression and classification tasks. It is a hierarchical model used in decision support that depicts decisions and their potential outcomes, incorporating chance events, resource expenses, and utility. This algorithmic model utilizes conditional control statements and is non-parametric, supervised learning. The tree structure is comprised of a root node, in which initially is contained all the population, that will start splitting, branches, that are the sub-trees, internal nodes, in which are placed the decision upon which the feature-based splitting is performed and leaf nodes, that are nodes that can't be split further, forming a hierarchical, top-down, tree-like structure. Very important is also the pruning operation, that cut down some nodes to avoid overfitting. The splitting operation may be based on different techniques, such as Information Gain (that aims to maximize the entropy reduction) or, for example, the Gini Index.
Also in this case, the differences with the other algorithms are in the ParamGrid implementation, that take in

input other parameters like the max depth of the tree, and in the Evaluation phase, where not only the whole tree built is printed, but also a Feature Importance operation is performed to retrieve the most influencing features for the regression, even if there is no Feature Selection operation performed, to avoid to remove too much noise and to avoid overfitting, to whom the Decision Tree algorithm is already sensible.

```
def decisionTreeRegression(label):
    try:
        dt = DecisionTreeRegressor(featuresCol = 'features', labelCol=label)

        dtParamGrid = (ParamGridBuilder()
                       .addGrid(dt.maxDepth, [2, 5, 10, 20, 30])
                       .addGrid(dt.maxBins, [10, 20, 40, 80, 100])
                       .build())

        dtEvaluator = RegressionEvaluator(predictionCol="prediction", labelCol=label, metricName="rmse")

        # Create 5-fold CrossValidator
        dtcv = CrossValidator(estimator = dt,
                              estimatorParamMaps = dtParamGrid,
                              evaluator = dtEvaluator,
                              numFolds = 5,
                              parallelism=3)
        return dtcv, dtEvaluator
    except Exception as e:
        print(traceback.format_exc())
        print(e)
        print("Model Creation Error -> Cannot perform the creation of the model")
```

6.4.4 Gradient Boosting Tree Regression

Like bagging and boosting, Gradient Boosting is a methodology applied on top of another machine learning algorithm, that works upgrading a weaker model, in this case the Decision Tree (so it can be utilized for both classification and regression problem), with a stronger one. The algorithm starts by fitting a simple model to the data, such as a Decision Tree with one or two levels. The residuals from this model are then used to train a second model, which is added to the ensemble. This process is repeated many times, with each new model trained on the residuals of the previous models. The final predictor is the sum of all the models in the ensemble. Also in this case, the only differences are in the ParamGrid implementation.

```
def gbtRegression(label):
    try:
        gbt = GBTRegressor(featuresCol = 'features', labelCol = label)

        gbtParamGrid = (ParamGridBuilder()
                .addGrid(gbt.maxDepth, [2, 5, 10])
                .addGrid(gbt.maxBins, [10, 20, 40])
                .addGrid(gbt.maxIter, [5, 10, 20])
                .build())

        gbtEvaluator = RegressionEvaluator(predictionCol="prediction", labelCol=label, metricName="rmse")

        # Create 5-fold CrossValidator
        gbtcv = CrossValidator(estimator = gbt,
                               estimatorParamMaps = gbtParamGrid,
                               evaluator = gbtEvaluator,
                               numFolds = 5)
        return gbtcv, gbtEvaluator
    except Exception as e:
        print(traceback.format_exc())
        print(e)
        print("Model Creation Error -> Cannot perform the creation of the model")
```

6.4.5 Random Forest Regression

Also in this case, is utilized a technique, specifically bagging, through which an ensemble of weaker algorithms, in this case again the Decision Tree, are grouped together to achieve better performances and to make up for the weak points, such as the sensibility to outliers and overfitting. The bagging technique implements two operation: aggregation, based on the fact that every decision tree has high variance, but when all of them together are combined in parallel, then the resultant variance is low, as each decision tree gets perfectly trained on that particular sample data, and hence the output doesn't depend on one decision tree but on multiple decision trees and so, in the case of a regression problem, the final output is the mean of all the outputs; bootstrap, in which row sampling and feature sampling are randomly executed from the dataset, forming sample datasets for every weaker model. Also in this final case, the ParamGrid entity differ from the other algorithms, and in the CrossValidator entity, it has not been possible to use a number of folds greater than 2 due to memory resource limitations on the machine.

```
def randomForestRegression(label):
    try:
        rf = RandomForestRegressor(featuresCol='features', labelCol=label)

        rfEvaluator = RegressionEvaluator(labelCol=label, predictionCol="prediction", metricName="rmse")

        rfParamGrid = (ParamGridBuilder()
                       .addGrid(rf.maxDepth, [10, 20, 30])  # , 30
                       .addGrid(rf.maxBins, [40, 80, 100])  # , 80, 100
                       .addGrid(rf.numTrees, [50, 100, 200])  # , 100, 300
                       .build())

        # Create 5-fold CrossValidator
        rfcv = CrossValidator(estimator=rf,
                              estimatorParamMaps=rfParamGrid,
                              evaluator=rfEvaluator,
                              numFolds=2,
                              parallelism=2)
        return rfcv, rfEvaluator
    except Exception as e:
        print(traceback.format_exc())
        print(e)
        print("Model Creation Error -> Cannot perform the creation of the model")
```

6.4.6 Results and comparisons

In the following table, results about a full run of the application are showed. The tables compare the real values and the predicted ones for the label utilized for this test, that is the "temperature" and the RMSE for, every algorithm (also the CrossValidator parameters are the one showed in the previous figures and are the same for every algorithm, except for the Random Forest, due, as said before, to limited memory resources.
As expected, Linear and Isotonic from one side, and Decision Tree, GBTree and Random Forest on the other, have similar results regarding the RMSE. The algorithm who performed better is the Random Forest, even if maybe could have reached better results if it was possible to set the number of folds of the CrossValidator to 3, like the other algorithms, followed by the other Tree-based algorithms, while the worst algorithm is the Linear Regression one. Regarding the prediction based on the reading received through Kafka, however, it's plain to see that every algorithm predicted a value in line with the real ones.

Real Values	Linear
33.98	31.25407
34.15	31.38908
37.18	357766
36.29	33.7722
36.17	30.7569
RMSE	1.2866

Table 16: *Table Comparison Linear Regression*

Real Values	Isotonic
34.23	34.9828
33.98	34.9828
34.68	34.9828
35.80	34.6823
36.17	34.6823
RMSE	0.9449

Table 17: *Table Comparison Isotonic*

Real Values	Random Forest
37.18	35.3412
33.58	33.3794
34.21	33.2916
31.59	32.1981
34.76	34.7337
RMSE	0.2642

Table 18: *Table Comparison Random Forest*

Real Values	GBTree
33.01	33.3885
34.15	33.4693
37.18	35.8182
34.68	33.4718
34.33	33.4693
RMSE	0.5414

Table 19: *Table Comparison GB Tree*

Real Values	Decision Tree
34.23	34.2200
34.15	34.1500
35.40	34.9150
35.06	34.9150
33.65	33.0150
RMSE	0.3685

Table 20: *Table Decision Tree*

Algorithms	Real Value: 33.03	RMSE
Linear	35.3338	2.3038
Isotonic	34.5801	1.5501
DecisionTree	33.6952	0.6652
GBTree	34.0936	1.0636
RandomForest	33.3613	0.3313

Table 21: *Comparison between real and predicted values and RMSE.*

7 Conclusion

Weka gave us the opportunity to apply classification and clustering algorithms to the provided dataset. Through these analyzes and the results that we deemed valid, we were able to continue the analysis work with the dataset we created. Node-RED has proven to be an helpful solution when dealing with linking together different sources and sinks for data collection, since we've been able to connect together every part of the system, like weather API, Arduino board, Google Drive and Apache Kafka. Considering that the process is also lightweight, we think it's the best solution to be executed on a constrained device like a Raspberry Pi.
Regarding the machine learning implementation, as seen previously, the results are very encouraging, also given the fact that certainly the size of the dataset utilized isn't as big as it should, and that for sure not every possible optimization has been applied. Nonetheless, a big improvement that is possible to apply and that is relatively easy to reach is to deploy the Spark application on a cluster, to fully exploit the parallelism and the memory resources, that not only can give a boost in term of execution speed but can also allow to implement more complex optimization techniques and to process bigger volume of data.