POLITECNICO DI TORINO

Collegio di Ingegneria Informatica, del Cinema e Meccatronica
Classe LM-32
Corso di Laurea Magistrale in
INGEGNERIA INFORMATICA (COMPUTER ENGINEERING)

Design and development of a full-stack platform for validation of the WRF weather forecasts model

Candidato:

Marco Bergesio

Relatore:

Prof. Daniele TRINCHERO

Correlatori:

Giovanni Colucci

Massimiliano Manfrin

Federico Spanna

Andreoli Valentina

Anno Accademico 2019 - 2020

ABSTRACT

Background

La tecnologia diventa giorno dopo giorno sempre più pervasiva in ogni ambito, anche in quelli apparentemente da lei più distanti, con lo scopo di cercare di migliorare l'attività dei settori storicamente non tecnologici, quali ad esempio l'agricoltura. Proprio in questo campo, oggigiorno sono disponibili molti modelli matematici che simulano la crescita delle piante e lo sviluppo delle loro malattie. Tali modelli operano con dati raccolti in tempo reale da appositi sensori posizionati nei campi coltivati. Grazie a questi sensori e all'attività di analisi dei dati è possibile prevenire l'insorgere delle malattie agendo in anticipo con opportune contromisure. Contemporaneamente, sono disponibili molte strategie per il calcolo di previsioni meteorologiche che potrebbero essere di supporto a tali modelli, ma questi due mondi che condividono un terreno comune non sono in comunicazione tra loro.

Obiettivo

Questa tesi vuole mettere in contatto il mondo della meteorologia con quello dell'agrometeorologia, introducendo l'utilizzo delle previsioni meteorologiche nell'applicazione dei modelli agrometeorologici. Se fossero disponibili non solo dati raccolti in tempo reale, ma anche dati di tipo previsionale, i modelli potrebbero produrre risultati migliori. Questo renderebbe possibile intervenire con ancora più anticipo per prevenire le malattie delle piante causate dalle condizioni meteorologiche. Ma prima di poter introdurre l'utilizzo delle previsioni meteorologiche nell'applicazione dei modelli agrometeorologici, è obbligatoria una fase di validazione dei dati previsionali per valutarne la bontà e il grado di affidabilità. Questa tesi fornisce il punto di partenza per effettuare tali operazioni di validazione dei dati previsionali.

Metodologia

Per raggiungere l'obiettivo descritto, si propone la realizzazione di una piattaforma web che raccolga i dati previsionali, forniti dal Dipartimento di Fisica dell'Università degli

studi di Torino, e i dati misurati dalle stazioni agrometeorologiche della Regione Piemonte. Questi dati sono debitamente organizzati e memorizzati in un database. Per permettere le operazioni volte a validare i dati, l'applicazione fornisce un'architettura a plugin, in grado di permettere in modo facile e veloce l'aggiunta di nuove funzionalità in futuro.

Conclusioni

Questa applicazione permette di validare le previsioni meteorologiche e di effettuare miglioramenti nel loro calcolo se necessari. Se da questa analisi risulterà possibile utilizzare le previsioni meteorologiche nell'impiego dei modelli matematici fitosanitari, si apriranno nuovi ed interessanti scenari per quanto riguarda il mondo dell'agricoltura, che potrà così sempre più soddisfare le sentite esigenze di un approccio ecosostenibile volto alla salvaguardia dell'ambiente, limitando l'utilizzo delle risorse non rinnovabili e la produzione di sostanze inquinanti. L'impiego della tecnologia in tale settore permette di anticipare il verificarsi degli eventi e quindi di agire in modo opportuno per limitarne i danni che si verificheranno, utilizzando solamente le risorse necessarie e niente più.

ABSTRACT

Background

Technology becomes more and more pervasive day after day in every area, even in those apparently more distant from it, with the aim of trying to improve the activity of historically non-technological sectors, such as agriculture. Precisely in this field, many mathematical models are available today that simulate the growth of plants and the development of their diseases. These models work with data collected in real time by special sensors located in the cultivated fields. Thanks to these sensors and data analysis it is possible to prevent the onset of diseases by acting in advance with appropriate countermeasures. At the same time, many ways are available for calculating weather forecasts that could support these models, but these two worlds that share common aspects are not in communication with each other.

Goal

This thesis wants to connect the agrometeorological world with the meteorology one, introducing weather forecast in application of agronomic models. If not only real-time collected data were available, but also weather forecast ones, models could produce better results. This would make it possible to act even earlier to prevent plant diseases caused by weather conditions. But before being able to introduce the use of meteorological forecasts in the application of agrometeorological models, a validation phase of the forecast data is required to assess its goodness and degree of reliability. This thesis provides the starting point for performing these validation operations of the forecast data.

Methods

To achieve the described target, we propose the creation of a web platform that collects forecast data, provided by the Physics Department of the University of Turin, and data measured by the agrometeorological stations of the Piedmont Region. These data are duly organized and stored in a database. To allow operations aimed at validating data,

application provides a plug-in architecture, capable of allowing the addition of new features in the future quickly and easily.

Conclusions

This application allows validation of weather forecasts and correction and improvement in their calculation if necessary. If from this analysis it will be possible to apply meteorological forecasts in the use of phytosanitary mathematical models, new and interesting scenarios will open up regarding the world of agriculture, which will thus increasingly be able to satisfy the felt needs of an eco-sustainable approach aimed at safeguarding the environment, limiting the use of non-renewable resources and the production of polluting substances. The use of technology in this sector allows to anticipate the occurrence of events and therefore to act appropriately to limit the damage that will occur, using only the necessary resources and nothing more.

Index

1 Introduction

Meteorology is a branch of the atmospheric sciences which is focused on study atmospheric chemistry and atmospheric physics, with a particular regard on weather forecasting. Weather forecasting is the application of all scientific and technological knowledge to predict the atmosphere condition for a given location and a given time. Weather forecasting have a scientific approach since the 19th century. To produce weather forecast, a large quantity of data about the current atmosphere situation has to be collected.

Agrometeorology is the application of meteorological knowledge in agricultural production and keeps into account relationship between atmosphere, soil and vegetation. Agrometeorology is fundamental to choose which type of production grows in a particular area and what improvements can be adopted depending on the weather conditions.

It is clear that meteorology and agrometeorology have a lot of things in common, but they do not communicate so much. They have a shared starting point which is the use of weather forecast collected data, but they proceed in different directions, each one doing its own stuff. It is also clear that agrometeorology could get a big help from meteorology, because nowadays a large number of agrometeorological mathematical models are available that are able to simulate phenological phases of plants and also of plants disease. Until now, all these models worked on historical data or, at most, with real-time data. In the agronomic field, models provide, for example, the daily simulation of growth, development and productivity of many plant species, as well as the use of the water content of the soils, and also the simulations on the effects of diversified fertilizations on crop yield in the various stages of vegetable development. In the phytosanitary[1] field, the knowledge of the relationships between the dynamics of climatic factors and the biological and phenological behaviour of the organisms constituting the agricultural biotic complex is an element of great importance for the

[1] Phytosanitary: relating to the health of plants

improvement and rationalization of crop management. In the entomological[2] field, categories of mechanistic models have been developed that can numerically link environmental variables to the development of specific insect resources.

At an operational level, some regional authorities have started to adopt the most tested modelling solutions on a permanent basis, in order to respond to requests coming directly from the agricultural world (data supply, alert and forecasting services). At the same time, the recent regulations issued by the European Community on plant health protection and weed control attribute an important role to agrometeorological supports and to the use of modelling tools as fundamental supports for integrated crop management. In this point of view, agrometeorological modelling can be a great help because current knowledge allows to simulate, estimate and reconstruct a large number of variables of fundamental importance, which can be used both directly to provide climate information and knowledge of the territory, and indirectly as guide for the application of more complex modelling tools linked to soil-environment-cultivation interaction.

If to all of this is given a predictive meaning and not just a diagnostic one, the value of modelling applications increases exponentially. The possibility of obtaining numerical climatic forecast information of a medium-term, to be subjected to a validation process using data actually measured, would allow the development of very useful operational supports in the area, making possible planning the correct crop management interventions. All this therefore requires both research in the modelling field (validation and processing) and a technical-operational support system in order to make this big amount of information accessible to users.

This thesis wants to go in that direction: it wants to try to connect the two unlinked worlds, meteorology and agrometeorology. But before integrate weather forecast in agronomic modelling it is essential to conduct a rigorous and precise study in order to understand how accurate weather forecast are. This step is fundamental because weather forecasts are strongly influenced by the geographical territory in which are

[2] Entomology is the branch of zoology concerned with the study of insects.

computed. It is not rare that even adopting the same weather forecast model we obtain completely different results depending on the physics features land. If we find the specific way in which weather forecast models act in the geographical land of interest, it would be possible to apply some corrections in order to limit intrinsic errors, obtaining a more precise and reliable forecast value. This is the goal of this job: we want to realize a platform that works as starting point, making easy to perform all needed validation.

The activity of acquiring data from meteorological stations and meteorological forecasts and the organization of such data intends to create the starting point for building a system that supports agricultural activities. The aim is to optimize agronomic practices in order to manage water resources, chemicals products for agriculture and fertilizers more and more carefully and rationally. The integrated system of data acquisition on the territory (through field monitoring and meteorological stations), the elaboration and diffusion of technical information on the territory will allow to achieve this objective.

We realized a collaboration between Politecnico di Torino, the research group in Atmospheric Physics and Meteorology of the Physics Department of the University of Turin which provides weather forecast information using the WRF model and the Agrometeorological Section of the Piedmont Council which provides measured data.

2 WRF model

Weather forecasts used in this thesis are computed using the WRF (Weather Research and Forecast) model, version 4.1.2.

The Weather Research and Forecasting (WRF) Model is an atmospheric model developed for numerical weather prediction (NWP) and for research [1]. The effort to develop WRF began in the latter 1990's and was a collaborative partnership of many entities, like the National Center for Atmospheric Research (NCAR), a US federally funded research and development center focused on meteorology, climate science, atmospheric chemistry, solar-terrestrial interactions, environmental and societal impacts, the U.S. Air Force, the Naval Research Laboratory, the University of Oklahoma, and the Federal Aviation Administration (FAA).

Since its first release on December 2000 WRF has become very popular not only among scientific community in US: nowadays it is the most used atmospheric model all over the world and it has a large worldwide community of registered users (a cumulative total of over 48,000 in over 160 countries [2]). Figure 1 shows a graphical representation of WRF growth starting from 2000 up to now.

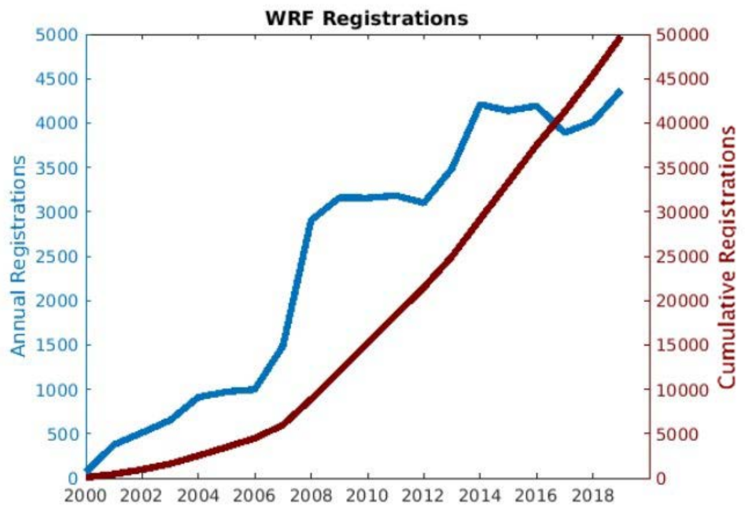


Figure 1: WRF Registration from 2000 to 2020

2.1 Background

During the 1990s the fifth-generation Pennsylvania State University and NCAR Mesoscale Model (MM5) was very popular in the research community because it was able to work with smaller atmospheric scale and because it could run on workstation-level computer. However, even if that model was nonhydrostatic[3], it was not optimized to work with such small areas, it was not accurate and it lacked a framework helping the addition of advanced physic parameters or that would support many desirable software attributes, like portability, parallelism, extensibility and APIs (Application Programming Interface). Those were the main causes that lead to start developing a new model, the WRF.

Compared to the previous models, such as the MM5, what emerged was a superior in high-order numerical accuracy and scalar conservation properties. Moreover, an innovative framework for the model's dynamics, physics and input-output components was added. This software architecture united the modelling components logically and efficiently, allowing system extensibility, ease of development, and scalable massively-parallel operation. Another development attempt addressed the pre-processors for domain and input preparation while a separate effort was related to data assimilation. Working on these task leads to the release of the first WRF model in December 2000.

WRF is officially supported by NCAR, which also provides regular workshops and tutorials on it. WRF's widespread acceptance is in part due to its being provided without cost, copyright encumbrance, or restrictions on modification. Though WRF is mature, it continues to advance. The system is being vigorously applied in new research directions, real-time settings, and marketplace opportunities.

Starting from its releasing, WRF has grown during the time, and in July 2018 was released the 4th version, also known as ARW (Advanced Research WRF), more flexible and updated to the last features, making it more and more used above all in climate research. Indeed, its potentiality in resolving smaller-scale atmospheric and land surface

[3] In LAM (Limited Area Model) there are two main categories: hydrostatic and nonhydrostatic models. The latter are the vanguard of meteorological modelling and they are capable to works with smaller areas.

processes exceeds the traditional global models used for climate projections in particularly for regional areas.

2.2 WRF and its applications

WRF produces atmospheric simulations. The process consists of two phases: the first one is about configuration of the model domain(s), ingest the input data, and prepare the initial conditions, and the second one is the execution of the forecasting model. The forecast model components operate within WRF's software framework, which takes care of manage the I/O operations and the parallel-computing communications.

WRF simulation starts with the WRF Pre-processing System (WPS). First of all, WPS inserts all geographical information, such as the topography of the land use, to set up the user's model domains. Next, it takes, reformats, and interpolates the required atmospheric data to the user's domains. Lastly, the input fields are put on the model's vertical levels and lateral boundary conditions are generated. At this point WRF is ready to run.

By design, WRF can also be configured to perform idealized simulations. This specific aspect allows users to study processes in a simplified setting by varying some parameters and initial conditions. WRF currently comes with 12 different idealized scenarios, such as flow over topography, supercell convection and tropical cyclones. Moreover, each user can construct other different idealized configurations.

WRF can also be executed as a global model on a latitude-longitude grid.

WRF has been extensively used for research with both real data and idealized configurations. In recent years the use of WRF for research on the regional climate has increased and, in this, the strength of WRF is solving the atmospheric and terrestrial surface processes on a smaller scale better than the global models traditionally used for climate projections.

The WRF Data Assimilation System (WRFDA) is the main system for collecting data for WRF. It can handle three-dimensional and four-dimensional variational (3DVAR, 4DVAR) approaches. In this way it can get a wide range of direct and indirect observation type, starting from traditional in place surface and over-air data to measurements coming from satellites.

Nowadays WRF is used operationally both from governmental centres and from private companies all around the world. The possibility to configure high-resolution domains, the variety of all possible input data, the flexibility in computation (especially when resources are limited) and the ability to adopt model advances from a global research community have made it very attractive for real-time forecasting. In the US, National Centers for Environmental Prediction (NCEP) uses WRF in support of the National Weather Service in a number of different applications, like energy, hydrology, severe weather, aviation weather and air quality. One of the recent custom developed versions of WRF is focus on simulate emissions and transport of smoke from wildfires. Also, the usage of WRF in private sector is very important. There are many different implementations, each one focused on different target: there are custom WRF that support flight forecasting to predict convection, turbulence and icing, other implementations are focused on optimizing energy production from wind and solar and other more are aimed at supporting agriculture.

2.3 Output

In this work, weather forecasts come from an accurate analysis made by the research group in Atmospheric Physics and Meteorology of the Physics Department of the University of Turin. The model computes weather forecasts in 2850 distinct points forming a rectangle area of 50 distinct values of latitude and 57 distinct values of longitude, as shown in Figure 2. The distance between one point and the other is equal to 0,0369 degrees, both in latitude and in longitude. Forecast values are to be intended as referred not only for the specific point, but for the area whose point is the center. For this reason, to better understand the whole forecasts data, is provided a new map in Figure 3 showing the point with its rectangle area of influence.

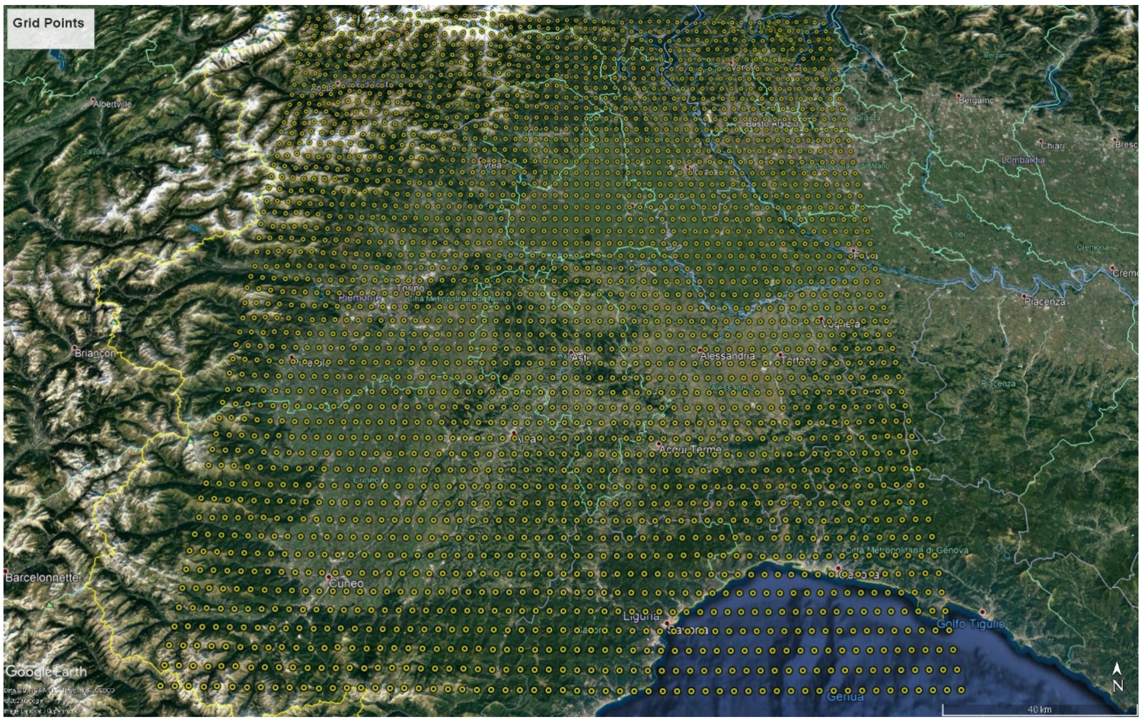


Figure 2: grid points from Google Earth

The model computes five-days weather forecast every six hours every day, starting from midnight, for every point in the area described above. Data are stored in Network Common Data Form (netCDF), that is a set of software libraries and machine-independent data formats that support creation, access, and sharing of array-oriented scientific data [3]. It is also a community standard for sharing scientific data. The major characteristics of netCDF files are:

- Self-description, because each file has information about data contained in the file itself
- Portability, because files can be accessed by computers with different ways of storing integers, characters and floating-point numbers
- Efficiency, because it is possible to perform direct access to a small subset of the entire dataset without having to process all the data stored in the file. This feature is very important because allows to save time and memory

- The possibility to append new data in an existing netCDF file without copying the entire dataset or redefining its structure
- The possibility to perform multiple accesses by one writer and many readers at the same time

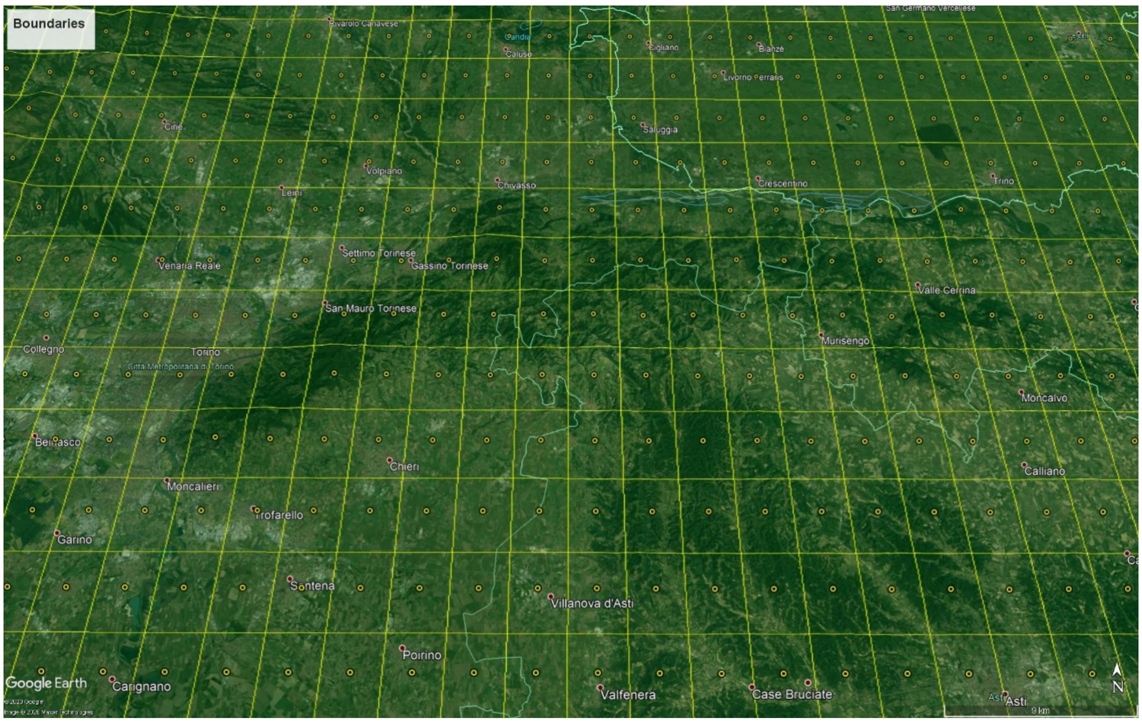


Figure 3: detail of WRF area with boundaries for each grid points from Google Earth

The model provides hourly data, but we compute a daily mean of them because Piedmont Council and the Physics Department of the University of Turin decided to work with daily values.

3 Weather stations

Measured data are provided by Piedmont Council’s weather stations.

Piedmont Council follows the trend of the meteorological situation through a monitoring system of the agricultural territory composed by a set of detection stations called RAM (Regional Agro-Meteorological network) [4]. This network has been working for many years and this led to the creation of a large database. This network is the starting point to setup agrometeorological reginal activities.

3.1 Background

Over the years, a new awareness has emerged: the management of production processes in agriculture must lead to a final result capable of associating the aspects of quantity and quality with those of sustainability and respect for the environment, taking also into account consumers and operators health.

The central core of this new trend is the environmental protection: agriculture has to dealt with climate changes, it has to adapter itself to new climate and, at the same time, it has to try to reduce pollution production responsible of the climate changes. Operating in this direction, Piedmont Council is acting to:

- Limit the contamination of non-renewable resources (water, soil, air) by agricultural activities
- Support the restoration, maintenance and improvement of natural and agricultural biodiversity and the landscape
- Improve the conservation of the soil system
- Spread agricultural practices suitable for increasing carbon sequestration
- Support water saving in the exercise of agricultural activities

Moreover, it supports and promotes more sustainable farming practices on the environment and consumer health. With regard to this last point, the Piedmont Region also wants to optimize the use of chemical products useful for fighting plant diseases.

In this context, innovative strategies and techniques have become established such as integrated, organic production or conservative agriculture[4]. To achieve the large-scale objectives that these techniques propose, however, it is necessary to overcome many operational and organizational obstacles. Among these, the need for farmers to benefit from technical and operational information to plan decisions regarding the best techniques to be adopted at a specific time of the season in order to obtain the best production result. European and national regulations also underline the fundamental importance of this type of support and underline that the Regions ensure the creation of monitoring networks and information distribution systems.

In this context, the evolution of the adopted methods and the availability of tools like diagnostic and forecasting modelling in the agrometeorological field has given a strong impetus to the increase of technological applications in agriculture. This sector has experienced a moment of strong rise in recent years all over the world and it is still in constant expansion, also due to the parallel growth of Information and Communication Technology (ICT), which has made possible to develop and implement increasingly complex systems. The agricultural sector needs to have detailed information available in terms of quantifiable numerical relationships relating to the correlation between climatic conditions and physiology of living beings, in order to have elements capable of explaining the development mechanisms of a living being in relation to variables that determine them. The models for estimating the water consumption of crops and for simulating the development and growth of crops, just to mention a few examples, constitute valid tools to support decisions to agronomic practices and agricultural management.

To achieve all these objectives, specific technical skills and technical support tools are required. What is need is:

- detailed meteorological data for the area of interest

[4] With *integrated production*, *organic production* and *conservative agricultural* are intended all those particular techniques that allow to use less chemistry products, less water and that produces less CO2 gas. In general term they are all those practises that keep into account the environmental protection.

- phenological and phytosanitary data provided by a monitoring network and, where available, by forecasting systems

The best way to collect all needed data is by using weather stations located on the sites of interest.

3.2 Weather station network

The network of Piedmont Region is composed of more than a hundred of electronic weather stations. Continuous operation over the years has allowed the database to be fed, which has now taken on a very significant historical reference value. Many stations have been operating continuously since the second half of the 90s. In any case, most of the stations now have a climatic historical series of over 10-15 years which is today extremely valuable for capillary climatological analysis and interpretation of the atmospheric, agricultural and phytosanitary phenomena of the present time compared to the past.

The electronic weather stations transmit information daily via GPRS to an operational center for the collection and storage of meteorological data. Data that reach the Operations Center are subjected to validation, possible reconstruction of missing data to do not have incorrect values. This information is then used for specific agrometeorological applications, also of a model nature like explained before, and sent to specific subjects.

Measured data used in this thesis are provided by Agrometeorological Section of Piedmont Council. In particular, we use data collected by 138 agrometeorological stations located throughout the region, as shown in Figure 4.

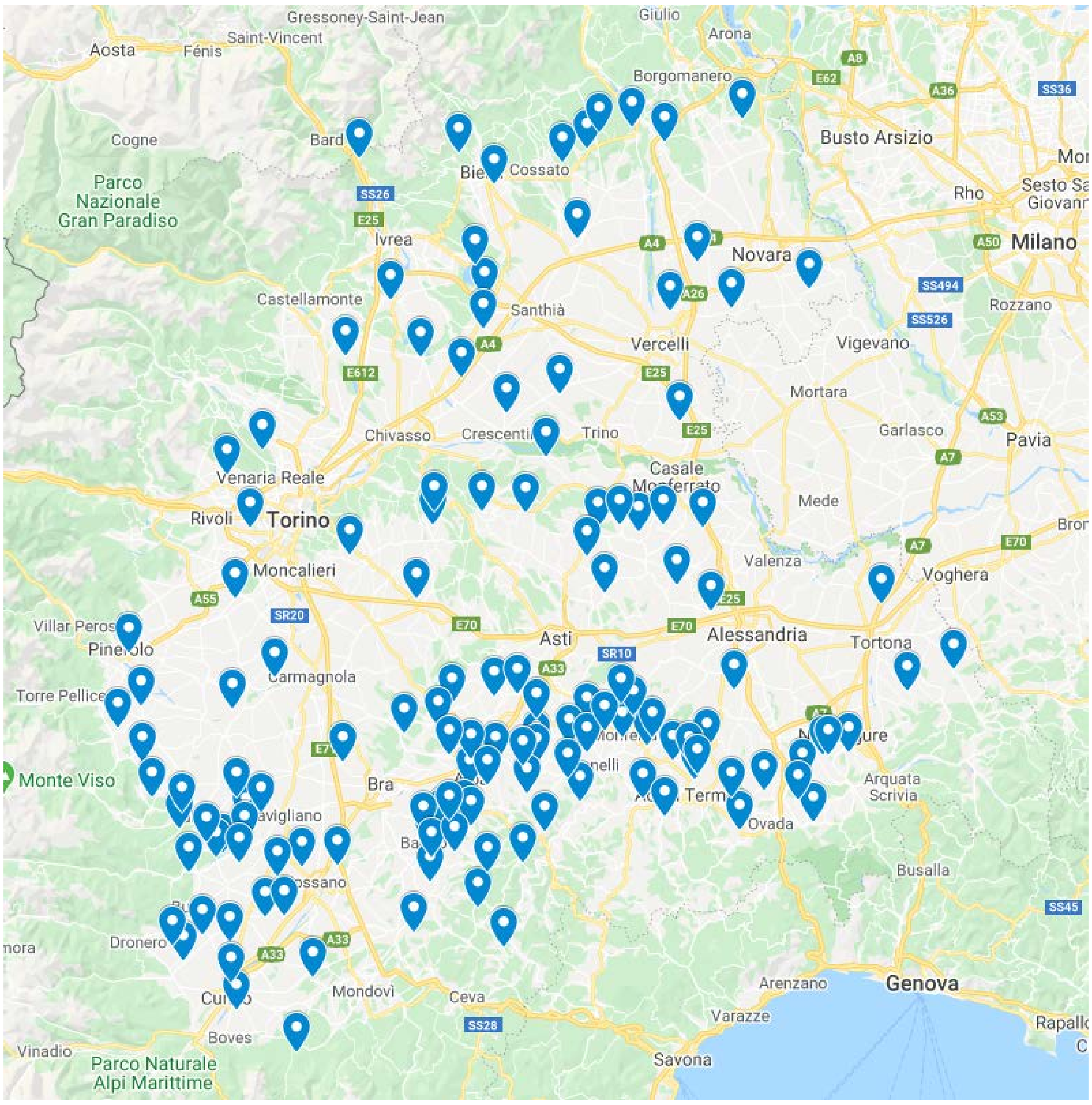


Figure 4: agrometeorological stations from Google Maps

3.3 Technical features

Stations forming this network are SIAP brand and they are able to acquire data on daily or hourly intervals. All stations record the following hourly quantities: average, minimum and maximum air temperature, average, minimum and maximum air humidity and rainfall. Moreover, some of them can optionally acquire other type of quantities, like atmospheric pressure, wind direction and speed, solar radiation, soil moisture and soil temperature.

The following table shows technical details for the most significant quantities measured by all stations involved in this work:

QUANTITY	RESOLUTION	ACCURACY	OPERATIVE RANGE
Air temperature	0.03°C	±0.3°C	-30°C ÷ +60°C
Air humidity	0.5% RH	±2% RH	0% ÷ 100% RH
Rainfall	0.2 mm	3% (0 ÷ 500 mm)	0 mm ÷ 999.8 mm

Table 1: technical details about measurements

Resolution is the smallest variation of the measuring value that can be measured by the instrument. Accuracy is the error between real and measured value. Operative range represents all possible measurable value.

Stations transmit collected data to a server using GPRS technology. In case of failure in radio connection, each station is able to save data for at least seven days, waiting for the restoring connection. Stations are powered by solar panels and they are able to keep on working for at least seven days in the absence of sun.

Moreover, the central server that collect data from all the stations provide spatial interpolation to reconstruct missing data. In order to validate the WRF model, we do not want to use those data so, for this reason, we get measured data before the interpolation procedure starts.

Data measured from weather stations are collected by 3A, a society born in 1997 focused on weather-climatic monitoring of agricultural environments [5]. It offers a support decision platform using data measured by weather stations and predictive modelling. We collect data from their server every morning at 7:00 AM, before they starting the interpolation procedure of missing data at 7:15 AM, performing a SOAP request.

4 Application overview

In order to allow validation of weather forecasts, the developed application, written in Python and using the Flask micro framework, collects forecasting data coming from the execution of the WRF model and measured data coming from weather stations. This application is cloud based and runs on a server hosted by the Electronics and Telecommunications Department of Politecnico di Torino.

The focus is without doubt the database, which plays a central role for the application purpose. Database contains data coming from weather stations and from the execution of the WRF model. All these data are properly organized into a certain number of table (more details in chapter 5). Moreover, they need to be elaborated in order to get some statistical information about the weather forecast accuracy.

To obtain measured data, application interacts with the Piedmont Council contracting company's server, which is implemented with the SOAP protocol and handles all data coming from Piedmont Council's weather stations. There is one dedicated module of the application which is responsible to execute this specific task and it is scheduled to be executed once every day. Contracting company's server exposes some API used by our application to retrieve information. SOAP (Simple Object Access Protocol) is an XML based protocol which allows machine-to-machine web interaction. This protocol defines messages format that applications can send in order to obtain or ask data. This protocol is platform independent and can be used by any programming language.

For what concerns handling forecasting data coming from WRF execution, there is another module of the application which is scheduled to be executed once every day and checks if new data are available in the DET storage. DET storage is a department storage developed with basic cloud technologies that allows to save and retrieve files. In the machine-to-machine interaction, DET storage uses WebDav protocol, which is an extension of the HTTP protocol and allows users to handle files on a remote server. For security reasons, WRF data are not directly saved on DET storage, because external accesses are not allowed. To solve this problem, WRF data are sent using rsync on a proxy server and subsequently they are stored in the DET storage.

The general application workflow is represented in Figure 5.

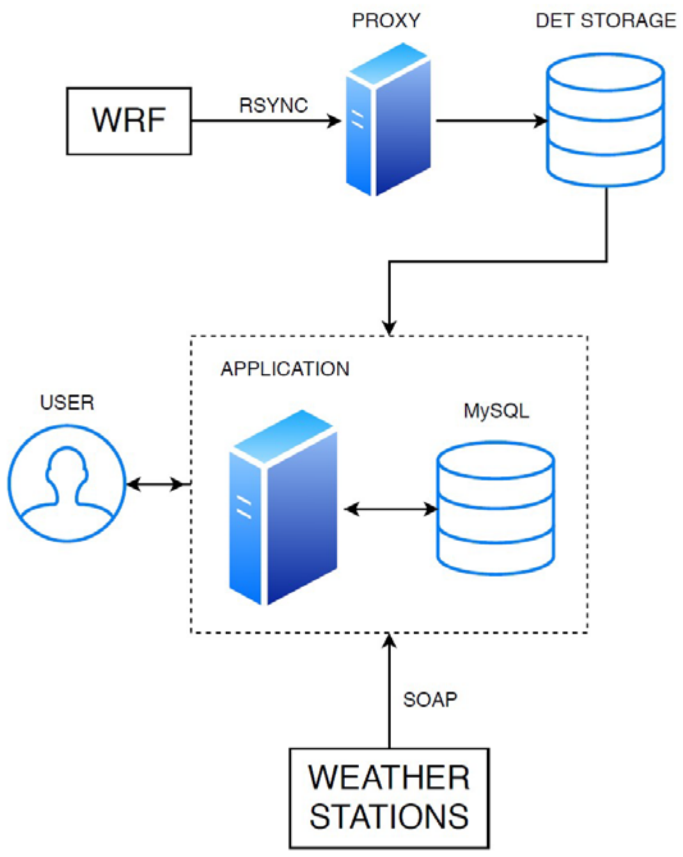


Figure 5: general application structure

Application also finds, for each weather station, the nearest weather forecast grid point in order to compare measured and forecast data for the same geographical point. This is done by using a function provided by MySQL which is able to compute the distance between two geographical points. This connection between a weather station and the grid point is not fixed: user can change the default grid point for one weather station by choosing another one, because the best weather forecast grid point for a weather station is not always the nearest one. E.g., if a weather station is located over the only hill in a flat area, measured data are affected by the hills themselves, while weather forecast are influenced by the predominant flat area. In this situation, it would be better to choose another nearby grid point located in a mainly hilly area.

To ease the validation task, the application provides a plugin architecture: in this way it is possible to add new features in an easy and fast way. It is just necessary to provide a Python file implementing the validation logic and a html file to display the results. It is also provided an example plugin to show the correct implementation and integration. Plugins can be activated or disabled by the user and can work using data (both measured and forecasting) of one weather station in a period of time of one year (chosen by the user).

The entire application is developed in Python because nowadays the main libraries for data analysis are available in this language and because this language is the one that suits best for plugins development also by non-expert people.

Figure 6 shows the application's home page. There is a big map provided by Google in which each weather station is identified by a marker. Above the map there is a table which can be hided if user does not want to see it. This table is just a list of all weather stations with geographical information such as latitude or longitude. By clicking over a marker in the map user can navigate throw the specific weather station page.

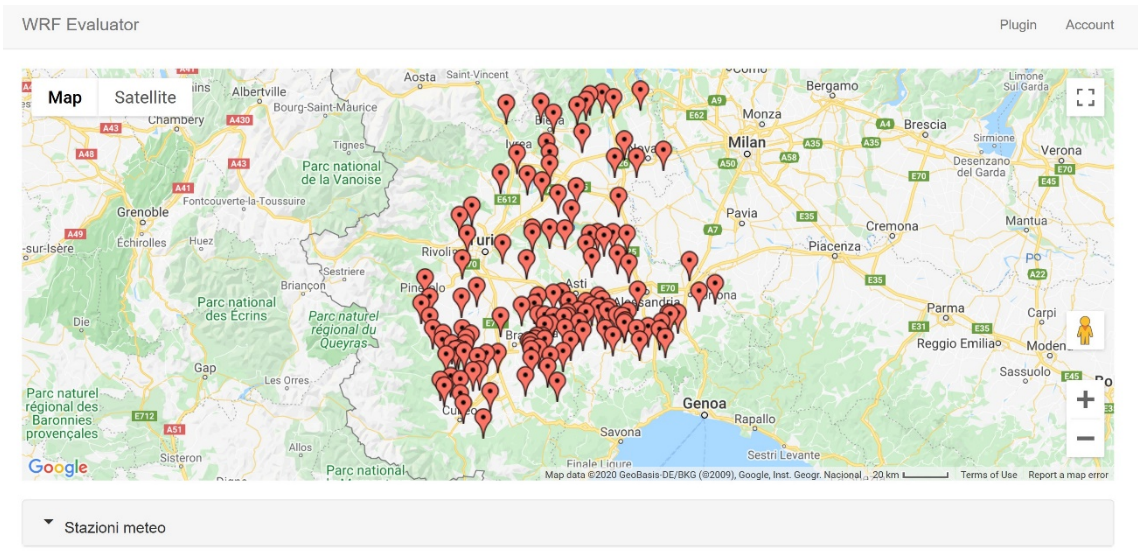
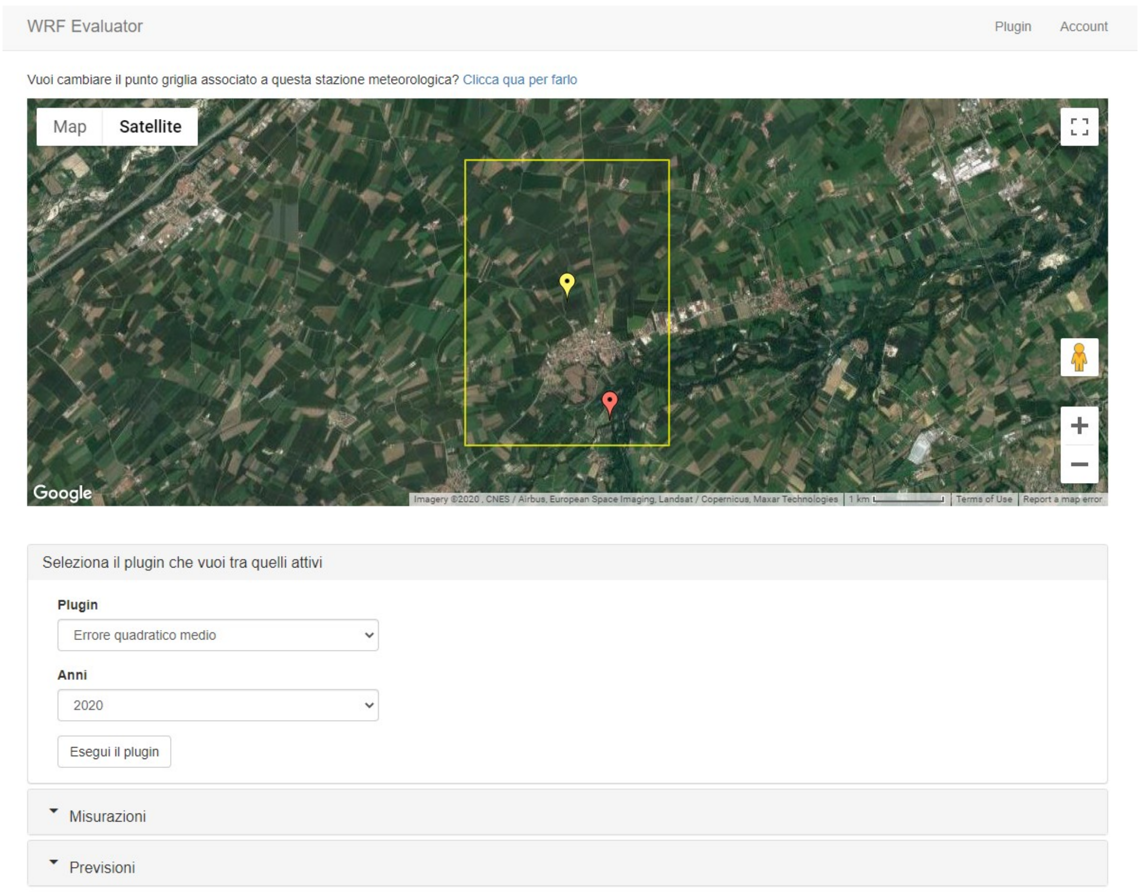


Figure 6: application home page

Figure 7 shows the page of one specific weather station. There is a map provided by Google in which there are one red marker for the weather station and a yellow one for the associated grid point. The yellow rectangle delimits the influenced area of the grid point. Below the map there is an area providing further information for the weather station. There are two hiding tables, one providing last five days available weather forecast and the other provides last five days available information about measured data. Moreover, user can execute one of the all available and active plugin. He can interact with the provided web form in which he can select one plugin and the year on which perform plugin. At the top of the page there is a link through which user can change the corresponding grid point of the weather station. To activate or disable plugin, user can navigate in the plugin page from the navigation bar in which are displayed all available plugin.


WRF Evaluator
Plugin
Account
Map
Satellite
Google
Imagery ©2020 , CNES / Airbus, European Space Imaging, Landsat / Copernicus, Maxar Technologies | 1 km | Terms of Use | Report a map error
Esegui il plugin
Misurazioni
Previsioni

Vuoi cambiare il punto griglia associato a questa stazione meteorologica? Clicca qua per farlo

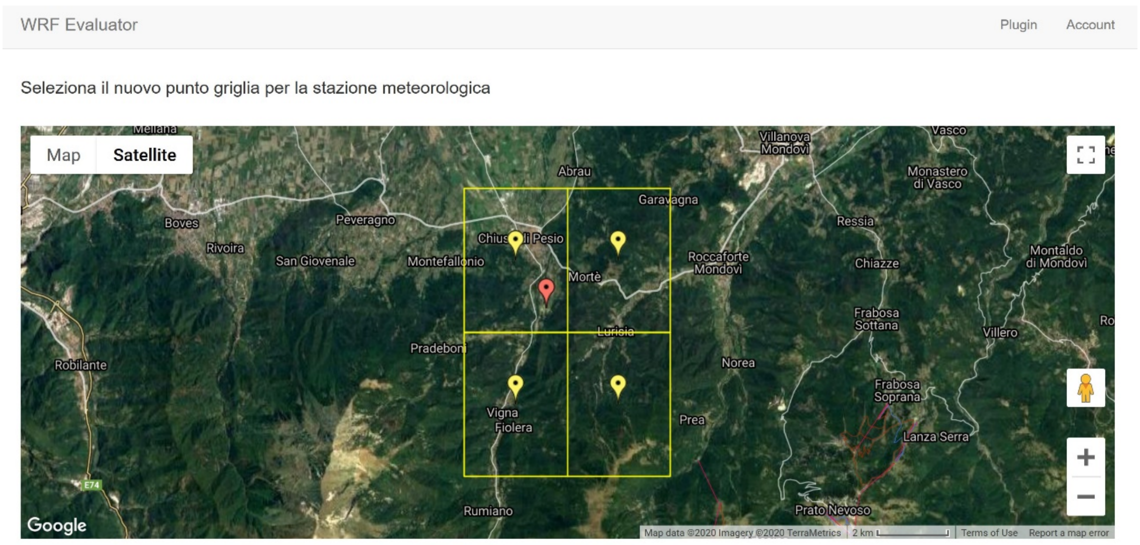
Seleziona il plugin che vuoi tra quelli attivi

Plugin: Errore quadratico medio

Anni: 2020

Figure 7: station's personal page

Figure 8 shows web page for changing the associated grid point for the weather station. The map, provided as usual by Google, shows the weather station identified by the red marker and the four nearest grid point. User can select the new grid point for that weather station by clicking on the yellow marker. In this way, application stores the new association and starts, from that moment, to compute weather forecast on the new grid point selected by user.



Seleziona il nuovo punto griglia per la stazione meteorologica

Figure 8: page for changing associated grid point

5 Technical features

5.1 Python

The whole application is developed in Python. We decide to choose this language because it is very flexible and robust and it has a lot of usage areas like web development, GUI desktop application, networking, database management and nowadays it is very popular among all applications concerning data analysis and numerical computations. There are some very famous libraries, like `Pandas` and `Numpy`, dedicated especially to this task that are widely distribute and usage.

Taking into account all plugin that will have to be developed in the future, I think that Python represents the best choice, because operations that need to be executed are oriented to the data analysis world. So, what explains the choice to choose Python is the desire to simplify the validation task that will have to be developed.

5.2 Flask

For what concerns application's backend, we used Flask framework. Flask is a micro web framework written in Python. The word "micro" is not to be intended in negative term, such as the application should be one file only or that it cannot be complex, but micro means Flask aims to keep the framework core as simple as possible, but extensible. There are a lot of things that Flask does not include, like a database: there is not a designed database to use within the framework, but every developer can choose the one that suits him best. This is just one example.

This is a specific choice of the framework in order to reach the biggest number of developers leaving them free to use Flask as they want, without any kind of imposition. Despite these apparently lack of the framework, there are a certain number of extensions covering the major aspects, not included in the core, that can be used like if they were part of the core itself. Flask was developed from the very beginning to be extensible. This is why extensions work very well with this framework.

Flask has three main dependencies:

- Routing, debugging and Web Server Gateway Interface (WSGI) subsystems which come from `Werkzeug`
- The template support which comes with `Jinja2`
- The command-line integration which comes from `Click`

All these dependencies are developed by the author of the framework, Armin Ronacher, and are automatically installed with Flask through the command `pip install flask`.

The entire application has been developed using PyCharm Professional which provides a good support in writing Flask application. Of course, we use a Python virtual environment, that is a copy of the Python interpreter in which we can install packages without affecting the global Python interpreter installed in the operative system.

5.2.1 Application factory pattern

Any Flask application is an instance of the `Flask` class to which web server passes all requests it receives from clients. The most quickly and easiest way to create an application is to create a global variable at the top of the code that instantiate the class, but this can be dangerous and difficult to manage as the application grows. So, for this reason, a good practise is to use the "application factory pattern", that consists in creating a function, usually called `create_app` in which we can instantiate the Flask class and in which we do all the configuration stuff. Everything concerning the creation of the application happens into this function. In this way we also simplify the testing phase, because we can create different instances of the application with different configuration parameters. Moreover, we do not have to manage a global variable (using global variable is never a good idea, if we can avoid them, we should do it).

The factory is defined in the `__init__.py` file of the application package and is recalled from the entry point of the Flask application, the file associated to the variable

FLASK_APP. This variable contains the Python file that is executed when we start our application. This is the only place in which the application still exists as a global instance.

5.2.2 Routes and view function

When clients send requests to web server, this one forwards them to the Flask application instance, that needs to know which Python function it has to execute in order to satisfy clients' requests. So, we need an association between different URLs and Python functions. This connection is called *routes*.

Routes can be defined using Python decorator `app.route` exposed by the application instance. Decorators are a standard feature offered by Python language. They are commonly used to register functions as handler functions that will be executed when a particular event occurs. They must be written just before the function we want to decorate. Route decorator takes as argument the relative URL to associate the Python function that decorates.

Decorated functions are called *view functions*. The return value of this function is the response the client receives. Response can be just a simple string with HTML tag or can be very complex and can contains a lot of things. Embedding response string with HTML tag in Python file is not a great idea, because it is very difficult to maintain. Solution for this problem are *templates*, that will be treated later.

Using Flask, it is possible to build *dynamic routes*, that are expressed with the following syntax: *'/static_part_of_URL/<dynamic_part>'*. Any URL matching the static part is mapped to the route and when the view function is invoked, it receives as argument the dynamic part of the URL. An example of dynamic routes in our application are all those routes for a specific page of a weather station, in which the dynamic part is the id of the station. In this way we can have personalized web page for each station associated to only one Python function. It would not be possible to have one Python function for each different weather station, this solution is not scalable and is impossible to maintain.

5.2.3 Configuration

Flask, and also many of its extensions, gives the developer some freedom in how do things and, for this reason, we have to tell the framework our preferences. We can do this by storing our decisions in a list of configuration variables and giving it to the framework.

There are several ways in which we can set all the configuration options. The most simple solution is to define variables in a dictionary form, in which key is the name of the configuration variable and the corresponding value is the specific configuration parameter we have chosen. Despite its simplicity, this is not a good way to work because we are putting together the creation of the application and the specific details about its configuration. Best practise suggests to keep this logically separate, because in this way we can better maintain the code in future and we can easily modify configuration aspects preserving application's core. Following this trend, we create a dedicated class in a separate Python file for the app configuration, called `Config` class. In this class are defined all the configuration variables.

Among all configuration variables, the most important one is SECRET_KEY, that is used not only from Flask framework, but also from many extensions. This variable is a cryptographic key, useful to generate signatures and tokens. For examples, Flask-WTF extension uses it to protect web form against the cross-site request forgery (CSRF)[5]. Everything is done automatically if we have the secret key value properly set and if we include in form template a hidden tag in which Flask-WTF insert the validation token.

The value of the secret key is an expression consisting of two terms joined by the `or` operator: the first one tries to retrieve the value from an environment variable. If that variable is not set, the second term of the secret key expression set the value. This practise allows developer to keep things simple in development phase because they can check that everything work properly by setting on the fly an easy value for the secret

[5] Cross-site request forgery is a malicious attack in which unauthorized commands are transmitted from a user trusted by the web application. This happens because the cracker can steal user's cookies and, using them, he personifies the real user. This attack exploits the trust of a website for a particular user.

key, but, at the same time, ensures high security level in production if the environment variables are well set.

When all the configuration variables are properly defined in the “Config class”, we just need to start Flask in order that it can load and apply all the configuration rules defined.

5.2.4 Blueprints

A blueprint is a logical structure representing one part of the whole application. A blueprint can contain element like route, template, form, view functions and static files. Blueprints are useful because almost all applications are composed of a certain number of different subsystems, like the one for authentication, another one for error handling, another more for the main purpose of application and so on. Blueprints help in keeping things logically separate, so we can have one blueprint for the authentication subsystem, another one for the error handling and so on. In this way we follow a modular approach in developing our application with all its benefits, like the ease in the maintenance phase and the possibility to reuse part of the code for other project.

In essence, a blueprint is nothing more than a python package. In the `__init__.`py file we create the blueprint and the other file of the package contain the logic of the subsystem. When a blueprint is created, it is in a dormant state, it can do nothing and it is not part of the whole application until it is not registered. To make a blueprint active it must be registered with the application. The registration is made just after the creation of the application. During the registration, all the elements that blueprint contains are passed to the application and become part of it. So, we can think of a blueprint as a sort of temporary storage for the application’s functionality.

Our project contains three blueprints, one for the authentication subsystem, one for the error handling and the last one for the core business of the application.

5.2.5 Templates

A view function has to generate a response to a client request. In many cases a single request triggers many changes in the application state and the view function is the place in which all these changes are generated. For examples we can think to a user that want to register a new account on a website. He types an email and a password and, pressing the submit button, he sends the information to the server. When server receive the client request, the view function is called to handle it. This view function needs to interact with database to check if data provided by user are valid or not and then it has to generate a response to send back to the client with a success or failure message. These two tasks are called *business logic* and *presentation logic* respectively.

Put together business and presentation logic is not a good practise, because it makes the code hard to understand and to maintain. Template help developers keep things separate. We can move presentation logic into templates to have a better organized code.

Templates are files containing text of a response with placeholder variables for those parts that will be known only in the context of a request. The process that replaces placeholders with actual values and returns the final response is called *rendering*. To render templates, Flask uses a powerful template engine called *Jinja2*.

All templates have to be stored in the *templates* subdirectory located inside the main application directory. This is the default place in which Flask looks for template files. View functions use the `render_template` function to call the Jinja2 template engine. This function takes the filename of the template as first argument. Any additional argument must be given in the form key-value and it will be made available to the template engine as actual value in order to replace placeholders.

In template file, placeholders are enclosed in double braces. Jinja2 recognizes any type of variable, even complex type like lists, dictionaries and objects. Variable can be modified with filters, which can be added after the variable name with a pipe character as separator, like the following: "*{{ variable | filter }}*". The `safe` filter is interesting to learn more. By default, Jinja2 escapes all variables for security purpose. But there are some cases in which we do not want to perform escape task because we want to display HTML tags stored in a variable. The most common case is when we want to display a table passed to the template as variable. The variable already contains all HTML tags necessary for well display table in browser. If these tags are deleted, table is not well represented in the client's browser. In this case, we have to use `safe` filter to avoid escaping operation. It is very important to never use `safe` filter on untrusted value, for example input text coming from client.

Jinja2 offers many control structures useful to modify the flow of the template depending on runtime context. We can use the `if-else` statement, the `for` loop, we can define *macros* that are similar to Python functions. Jinja2 also support template inheritance, which is similar to class inheritance in Python. This function is very useful to define some aspect that are common to all pages in our application, for example the navigation bar. Without template inheritance we should define in each template the navigation bar. This approach produce code difficult to maintain, because, when a change occurs, we have to modify the same thing in many different places. There is a very high probability to forget something and to have therefore an inconsistent website. With template inheritance we can define a base template in which we developed navigation bar and other shared stuff. In the base template we define also a block that will be overridden by derived templates in which each template can put his particular stuff to display. In this way all templates will have the shared parts, but everyone will contain its own specific stuff.

5.2.6 Bootstrap

Bootstrap is an open-source web browser framework from Twitter that help in providing a clean and attractive user interface compatible with all modern web browser, both desktop and mobile. Bootstrap is a client-side framework, so the server is not directly involved. Server just have to provide HTML responses that references Bootstrap's CSS and JavaScript file. All this is done, of course, in templates.

The primitive way to integrate Bootstrap with our application is to modify HTML code, applying all changes recommended by Bootstrap documentation. This approach is not the best one, we should do a lot of work to properly display web pages. Luckily there is a Flask extension that make things simpler and faster. This extension is called Flask-Bootstrap and we can install it throw the command `pip install flask-bootstrap`.

Once that we have initialized Flask-Bootstrap, a base template including all the Bootstrap file and general structure is available to our application. Using template inheritance provided by Jinja2, we can extend this base template in order to have all Bootstrap features. We extend Bootstrap base template in our base template, in order to make it available to all our templates. In this way we are creating our base template in the same way Flask-Bootstrap provides its own base template.

5.2.7 Error handling

When an error occurs, it is important not to show error details to user: we do not want that user can get information about our application internal structure, but we want, at the same time, that the error page has the same layout of all other pages. To do this, we have to create custom error pages. Flask provide a mechanism for an application to install its own error pages. To declare a custom error handler, we have to use the `errorhandler` decorator, which receives in input the error code it has to manage. The error function is a real view function. In our application we define view function for the two most common HTTP error, 404 e 500. These view functions return a second value after the template they have to render, that is the error code.

When an error occurs, it is important to register it into a log file. In this way we are able to collect all failures that are not Python exceptions but they may be still interest enough to save and to resolve in a second moment. For this reason, we implement a log file for the application. To do this we have to add a handler function to the application. For this new handler we use the `RotationFileHandler`, which is able to rotate the logs ensuring that log files do not grow too much when application runs for a long time. We limit the size file to 10KB and we keep only the last ten files to not use too much space on disk.

5.2.8 User authentication

Authentication is a fundamental part of the application because it allows to keep track of users and it allows to perform operations only to registered ones and not to everyone. The most common method of authentication requires user to provide an identification, like a username or the email address, and a secret only know by him, that is the password. This is not the most secure way to authenticate a user, because there are some problems related to the password usage. The most common one is that user write down password somewhere to not forget them, but in this way if someone else find the password can login in place of the real person. Another common problem is that the same password is valid for many different accesses: in this way, if a cracker steals the user's authentication credential when they are sent over the network, he can personify the real user. To avoid all these problems there are some other authentication techniques that use different technologies, like for examples one-time password (OTP) or challenge-response authentication with asymmetric cryptography. These methods are no doubt safer, but they are more difficult to implements and to use. It is reasonable to adopt them when we deal with money or with very import and secret data. This is not our case and, for this reason, we can use the username-password authentication method without problems.

Even when we use the simplest authentication method, we have to keep into account some things to make our application as secure as possible. One of them is related to information stored into database. If database contains passwords in plain-text, if an attacker is able to break into our server and access the database, he can read and steal all users' password. If this happens, the risk is bigger then what we can think: even if we do not store any sensitive information in our database, it is known that most user use the same password for many different web sites. For this reason, we must do all we can to avoid that an attacker could steal users' passwords. What we can do is not to save password in database as plain-text, but we have to store a *hash* of it. Moreover, we can add the *salt* to the password before compute the hash. The salt is a random string combined with the password useful to avoid what are called dictionary attack[6]. A password hashing function takes a password as input, add salt to it and then applies several one-way cryptographic transformations to it. The result is a new string completely different from the original password that has no known way to be transformed back into the original password. Password hashes can be verified in place of the real password because hash functions are repeatable and they can be computed very fast: given the same input (password and salt), the result is always the same.

Password hashing is a very complex task to be implemented from scratch without making errors, so it is recommended to not do this, but to use one of the many available and tested implementations. One of them is provided by Werkzeug, that comes with Flask by default. There are two helpful functions we can use for this task. One is `generate_password_hash`, which takes in input a plain-text password and return the password hash as a string that can be stored in the database. This function has two more input parameters, one is the hash function to use (default value is *sha256*), and another one is the salt length. To combine salt with password, this function use *hmac*[7]. The returned string looks like this: "method$salt$hash". In this way we have all information needed to check password's hash. We use this function when we want to add a new user in the database. Otherwise, when we want to check if user gives the correct password when he wants to login, we have to use the `check_password` function, which takes in input the password hash generated from the previous function and the plain-text password given by the user. This function computes hash of the new password using the same salt and hash function used by the `generate_password_hash` function: if they are equal, it returns True, otherwise it returns False.

[6] A dictionary attack consists in look into a list containing hash of most used password to find user password. If we modify password with a random string before compute hash, it is impossible to find it into a pre-calculated list.

[7] HMAC (keyed-hash message authentication code) is a specific type of message authentication code (MAC) that use a cryptography hash function and a secret cryptography key. This is a standardized function that avoid problem that can occurs when we manually combine the secret key and the data.

When user login the application, his authenticated state has to be saved in the user session[8], so that it is remembered as he navigates through different pages. Without doing this, when user goes into a new web page, he has to login again. This is not acceptable. Flask-login is a small extension that helps in managing this particular aspect of a user authentication system. We can install it through the command `pip install flask-login`. Flask-login works with the application's own `User` object. To be able to do this, Flask-login requires the application's `User` model to implement the following methods:

- `is_authenticated`, which returns True is the user has valid login credential or False otherwise
- `is_active`, which return True if the user is allowed to login or False otherwise. A False value indicates disabled accounts
- `is_anonymous`, which is False for all regular logged-in user
- `ged_id`, which returns a unique identifier fir the user

We can implement all these methods directly in the `User` model class, but Flask-login provide `UserMixin` class that has a default implementation of all these methods appropriate for most cases. All we have to do is to make our `User` class to inherit from the `UserMixin` class and to provide a function that Flask can use when it needs to load a user from the database given its identifier. This function is decorated with the `login_manager.user_loader` decorator.

Flask-login provides the `login_required` decorator with which we can make a page accessible only to logged in users. If a not authenticated user tries to access a protected route, Flask-login will intercept the request and redirect the user to the login page instead. To make a route protected, we have to add the `login_required` decorator after the `route` decorator. The order is very important because, when we dealt with multiple decorators, each one affects only those that are below it. If we act in this way, first we protect the view function and then we register it to the given URL. Reversing the

[8] Whit "session" we mean all interactions between browser and server until the browser is closed. With sessions we can keep track of all choices made by user, like for example the login.

order will produce a wrong result, because the `route` decorator will register the view function before the `login_required` decorator protect its, and so that URL will be visible to everyone.

When user want to login the application, he fills the login web form. Then the view function will check if the given credentials are valid and correct and, if they are, it is executed the `login_user` function, which comes with Flask-login. This function registers the user as logged in for the user session. This function takes as input the user and also the “remember me” Boolean option. If this option is set to False, the user session will expire when the browser is closed, otherwise it will survive.

To logout user, Flask-login `user_logout` function is called to remove and reset the user session.

Last thing to do is to display the login or the logout button in the navigation bar of our application depending on the circumstance. To distinguish if a user is logged in or not, we can use the `current_user` variable, defined by Flask-login and available to view functions and templates. This variable contains the current logged-in user or an anonymous object if the user is not logged in. Using the `is_authenticated` property described above on the `current_user` variable we can understand if we have to display the login or the logout button.

For what concerns user registration, we do not expose any public URL that allows this operation because we do not want that everyone can register on our application. So, when we want to add a new user, we do it by hand.

5.2.9 Web form and Flask-WTF

Without web form, information can flow only in one direction, from server to client. If client wants to send some information to the server, without web form he cannot do that. With HTML it is possible to create web forms in which users can enter information. Form is then submitted to the server by the client web browser, typically performing a POST operation. By default, Flask provides a support in handling web form, but working only in this way can become tedious and repetitive, it is not a smart approach. Like in other aspect, even for working with web form there is an appropriate Flask extension. In this case, the extension to use is call Flask-WTF which, as usual, can be installed with the command `pip install flask-wtf`.

Unlike many other extensions, Flask-WTF does not need to be initialized when we create the application instance, but it expects the application to have the secret key properly configured because this key is used by the extension to protect all forms against CSRF attack, as explained before. Flask-WTF generates security tokens for all forms and stores them in the user session, which is protected with cryptographic signatures generated starting from the secret key value.

With Flask-WTF, each web form is represented in the server by a class that inherits from the `FlaskForm` class. Class representing web form has one object for any webform field. Any fields can have one or more validators attached. A validator is a function that checks whether data submitted by the user is valid or not. Here are some default validators provided by Flask-WTF:

- `DataRequired` which checks if field contains data after type conversion
- `Email` which checks if the input is an email (it checks only the syntax without control if that email address exists or not)
- `EqualTo` which compares value of two fields, useful when we ask user to repeat some input data to be sure (like password in registration phase)
- `Optional` which allows empty input in the field
- `Regex` which validates the input using a regular expression
- `AnyOf` which checks if the input is one of a list of possible value
- `NoneOf` which checks if the input is none of a list of possible value
- `NumericRange` which checks if entered value is within a numeric range
- URL which checks if the input is a URL

It is also possible to define some custom validator for our webform, in addition to the default ones. To do that, we have to implement a function, in the class in which we define the web form, whose name matches the patter `validate_<fiel_name>`.

When those methods are provided, they are invoked automatically in additional to any regularly defined validators. With these methods we can implement custom and more complex controls on the input data.

Flask-Bootstrap extension provides a high-level helper function that renders Flask-WTF form using Bootstrap's predefined style with just one single call. To do this, we have to import in the template file the `bootstrap/wtf.html` and then using the `quik_form` method, which takes the actual form as argument. This method renders the form with also the hidden tag in which Flask-WTF puts the security token.

When we register a view function with the decorator `app.route`, we can add a list of methods that will trigger that function. If we don't provide any list, Flask will call that function only if a GET operation is performed on that URL. When we deal with form, we have to register the view function to be executed both with GET and POST operations, because when a client fills a webform, his browser sends it to the server performing a POST operation.

The view function has to create an instance of the class that implements the form. This class offers the method `validate_on_submit` which returns True when form is submitted and data are accepted by all the field validators. In all other cases this function returns False. In this way we can easily determinate if we have to render the webform or if have to process it. When a user navigates for the first time to the URL associated to the view function that handles the form, he performs a GET operation because he wants to visualize the web form on his browser. At this point there are no data that need to be processed, so `validate_on_subimt` will return False and the view function understand that it just has to render the webform, without doing nothing more. But when client presses the submit button, his browser performs a POST operation on the same URL used before to ask the rendering of the webform. This time, if data are correct, the `validate_on_subimt` function will return True and the view function can perform its operation over data.

After a user submit a webform, it would be nice if we give him some feedback about the performed operation. This could be a confirmation message, a warning or an error. A typical use case is when a user tries to authenticate himself but provides wrong credential. If we simply show again the login page without any message, he most likely will not understand what happened and maybe he can think that our service is not working well. It would be better if we tell him that provided credentials were wrong, so he can easily try again paying more attention. Flask includes this functionality by default in its core throw the `flash` function. This function takes as argument the string we want to display as message and must be invoked in the view function, before render the template. But calling this function is not enough to get message displayed; templates used by application need to render these messages. Best place to do this is in the *base template*, in order to have this functionality available in all our application templates. To do this, we call the `get_flashed_message` function which makes available to the template all flashed messages. By performing a `for` loop over all those messages, we can easily display all them. Messages retrieved by `get_flashed_message` will not be returned the next time this function is called, so every flash message is just display one time.

Last thing we have to pay attention while using webform is the pattern submit-refresh. If a user submits a webform and then press the refresh button, the browser will show an alert message asking confirmation before reload page and therefore submit the form again. This happens because the refresh button repeats last request sent by the browser. If user presses submit button, browser performs a POST operation and so refreshing the webpage will cause a second form submission with the alert message. Submit form multiple times can cause some problems and, for this reason, browser asks confirmation before doing this. Moreover, many users cannot understand the confirmation message provided by the browser. It is not a good practise for web application to leave a POST operation as the last request performed by a user. To act accordingly to best practises, we have to respond to a POST request with a *redirect* instead of a normal response, such for example render a template. A redirect is a special response containing a URL instead of a string with HTML tags or a template to render. When browser receives a redirect response, it performs a GET request to the URL provided by the redirect. This flow can last a few milliseconds more because browser has to perform one additional request, but in this way the last operation is always a GET even if the user press the submit button after filling a webform.

5.3 Database

Database represents the persistent layer of the application. Our application has to store different type of data: first of all, of course, information about registered user, but this is not the focus. The predominant part is the one concerned to weather forecast and geographical information. The aim of this work is to build up a system capable to save data coming from the WRF model and from the weather forecast stations of Piedmont Council in order to allow some operation over this data to validate weather forecast.

The first decision to make before starting to build up the database layer is to choose between SQL and NoSQL database. SQL means “Structured Query Language” so, as its name suggests, it is a language specific for structured data. We can say that data are structured when it’s easy to think them in form of a table, where columns represent the different attributes and rows, called entry, are the data itself. The table form is the core of the SQL database because these ones describe everything using tables where every object is one row of one specific table. For example, for what concerns the register users of our application, username and password are the attribute of each user, while the different value of username and password of each user are the entry of the table. NoSQL means instead “Not Only SQL”. They can represent both structured and non-structured data, like for example multimedia information (photos, videos or audios). There is not one database that is better under every aspect respect another one, but the choose between SQL and NoSQL depends on the purpose of the application and on the type of data that we have to manage. Another aspect to consider when we have to decide what is the technology to use is if the structure of our data will change as time goes or not. This because, before using an SQL database, we have to define the structure of the database itself: that means that we have to properly define all tables, all columns of each table and also the relations between different tables and different columns of different tables (one column can referred another column on another table). There is a not negligible amount of work to do before start using an SQL database. If the structure of our data changes at a certain time, we have to redefine the whole database schema. On the other hand, NoSQL database does not have a pre-designed and fixed structure, basically because they don't have table, but data are stored in documents. So, they are good if we handle not structured data or data whose structure is not always the same. If at a certain time we have to save a new attribute of some entity, we simply add the new attribute in the document, we do not have to do nothing more. The freedom given by NoSQL database is of course an advantage under certain aspect, however, the rigid structure of SQL database is not just an imposition, but can become a help in avoid error and in highlight coherences and integrity in our database.

In our application, as I said, we have to work with weather forecast information, a type of data that lends itself well to be represented in a table form. Moreover, the aim of the job is to perform validation on weather forecast data, and for this reason it is better if we work always with the same type of data. There is no sense to modify data structure from a moment on. These are the main aspects that led me to choose an SQL database.

To realize the database, I follow the Entity-Relationship (ER) model, which is the most popular model for realizing SQL database. Entity represents classes of object of real world (persons, things, events…) that have common properties and autonomous existence. An entity occurrence is an object of the class represented by the entity. Entities are characterized by attributes, which describe the entity's properties. Entities are identified by one or more unique attributes. These attributes are call identifier and can belong to the entity they describe or they can be composed by one or more attribute of the entity itself joined with one or more attribute of another entity. In the last case, the entity is described as weak if it cannot identify itself alone. Relation represents a logical connection between two or more entities. Relations are characterized by a value of cardinality, which describe the minimum and maximum number of a relation occurrence in which an occurrence of an entity can participate. The minimum can be 0 (optional participation) or 1 (mandatory participation), while the maximum can be 1 (at most one occurrence) or N (arbitrary number of occurrence). So, we can have binary relationship one-to-one, one-to-many or many-to-many.

To properly development an SQL database using ER-model we must to follow some step. Starting from application requirements, we have to do the *conceptual design* of the database, in which we translate requirements into entities and relationships and we describe entities with attributes. After doing this, we have to produce the logical design consisting in produce real table that will populate the database from the conceptual design.

The conceptual design consists also in understand properly all data and decide which data are useful and must be saved and which one are useless and must be discarded. In this first phase we collaborated closely with colleagues from the Piedmont Council and the Physics Department in order to take these decisions.

After the data comprehension, we keep on doing the conceptual design by identify all the entities involved. The first is the one that describe the WEATHER STATION. Each weather station is characterized by a name and a geographical position expressed in term of latitude and longitude. There is also information about the altitude and the interval of data acquisition. These two last aspects are not essential, but we decided to save them notwithstanding, to make the database more complete. We also add an id to each station to better identify them and to simplify join operations in the workflow. The list of all the weather stations involved in this project was provided by the Agrometeorological Section of Piedmont Council with an Excel file. The second entity involved is MEASUREMENTS. Each measure is characterized by the weather station that made it, by the date in which it was produced and by a list of parameters related to weather forecast, like rain, air temperature and so on. The third entity is GRID POINT, described by a geographical position expressed in latitude and longitude and by an id. The last entity is WEATHER FORECAST, characterized by the id of the grid point whose is referred, the date in which has been computed and the date to which that forecast is referred.

A schema of the conceptual design is reported in Figure 9.

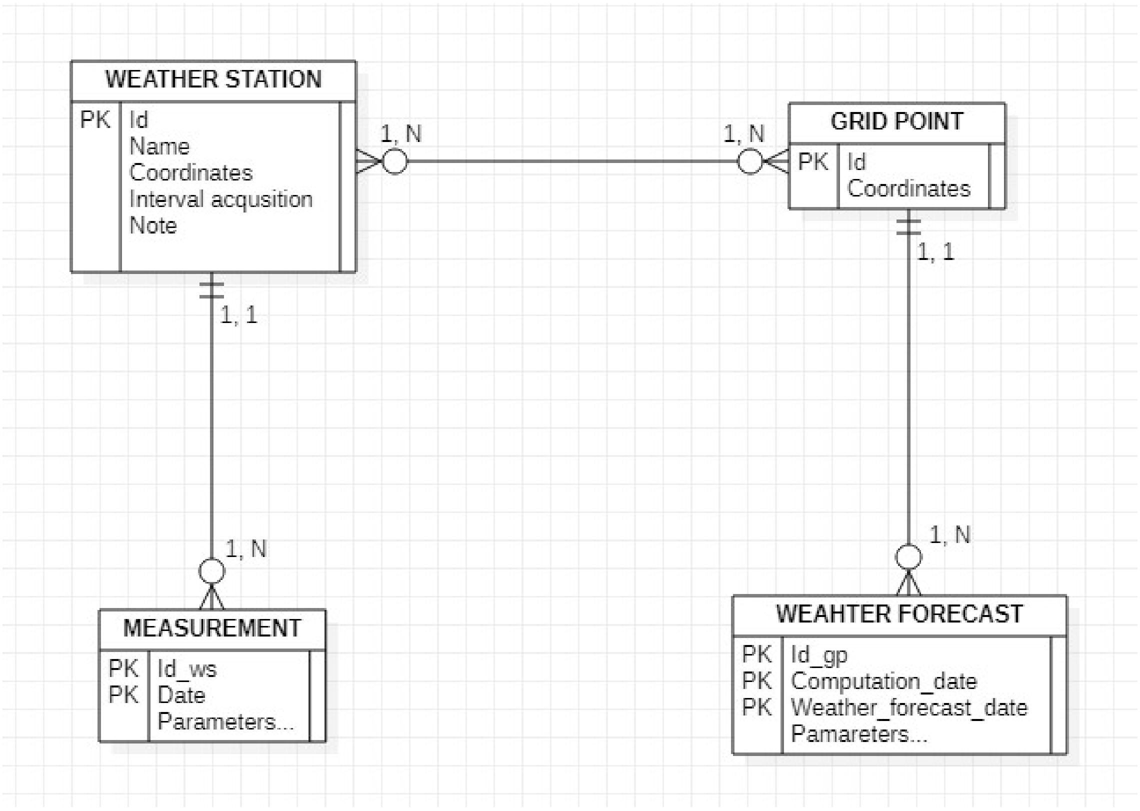


Figure 9: conceptual design of the database

As we can see, both “measurements” and “weather forecast” are weak entities because they are identified with an external identifier.

Relation between “weather forecast” and “measurement” is a one-to-many relation because for each weather station we have at least one measurement (we do not have in our database weather stations that not produce measurements), but each measurement is associated to one and only one weather station. The relation between “weather forecast” and “grid point” is also a one-to-many relation, because for each grid point we have at least one weather forecast and each weather forecast is associated to one and only one grid point. Relation between “weather station” and “grid point” is a many-to-many relation because each weather station can potentially be associated to many grid points and each grid point can be associated to many weather stations. This because our application finds, for each weather stations, the nearest grid point but gives user the freedom to change the associated grid point if he wants to perform other type of analysis. The user may want to change the related grid point because of geographical reasons: it can happen that one grid point covers a mostly flat area, but the weather station in located on a small hill belonging to the area referred by that grid point. Maybe, in situation like this one, user want to change the association weather station – grid point done automatically by the application considering just the distance choosing an area that better represents the geographical conditions in which the weather station is located. For that reason, the relation between weather stations and grid points is a many-to-many and not a one-to-many.

After doing this, we have to convert the conceptual model into the logical design. The goal of this step is to translate entities and relations into table. Translating entities into tables is quite simple in our case because we do not have redundancies, hierarchies, multivalued or composed attributes. So, for each entity we will have one table. On the other hand, translating relation is a bit more complex because we have to handle different type of relations and everyone has its own rules to apply. The many-to-many relation is translated by creating a new table that works as a bridge between the two entities involved. The primary key of this table is the combination of the primary keys of the two entities involved. To convert the two one-to-many relationships, we can proceed in two different ways: we can create a new table whose primary key is the one of the table which cardinality is equal to one or we can simply add the value of the primary key of the table with cardinality is equal to "many" in the table whose cardinality is equal to "one". To avoid creating two new tables, the second strategy has been adopted.

A schema of the logical design is shown in Figure 10.

Last step of logical design is to define referential integrity constraints. In particular, the "id_place" in measurements table references the "id_place" in weather stations table. This means that all the id present in measurements table must be present in weather station table. Other constraints like this one are the "id_place" of weather forecast table that references the "id_place" of grid points table, and the "id_weather_staion" and "id_grid_point" in the bridge table that reference respectively the id on weather station table and the id of the grid points table.

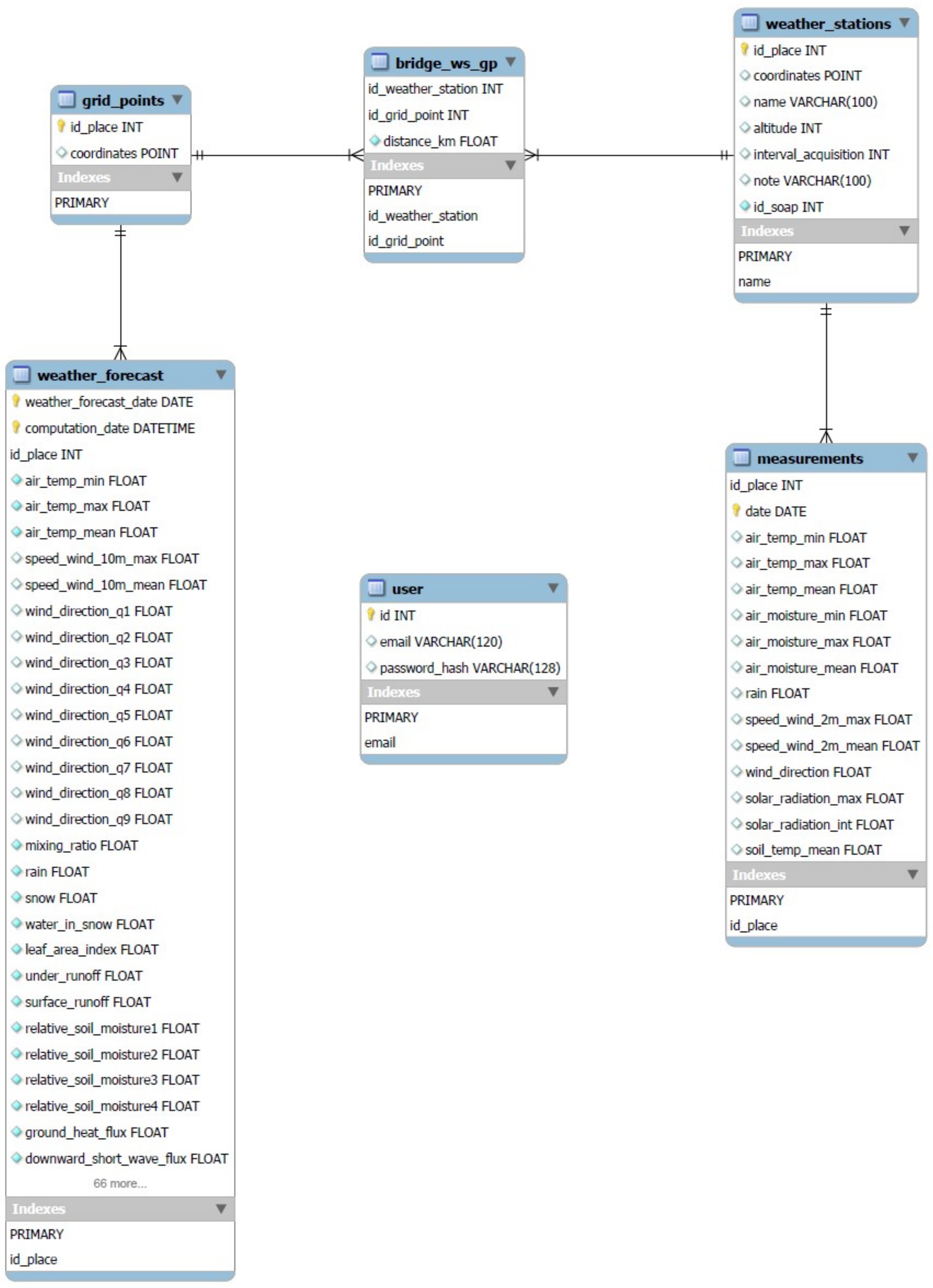


Figure 10: database schema

5.3.1 Database and DBMS

Database are nothing more than a set of files whose aim is to save permanently some data and making them available for any kind of software. Database must follow an internal structure, as discussed in the previous chapter of this thesis.

DBMS (Data Base Management System) is a software designed to store and manage database. Programs that use database cannot operate with database directly, but they interface with a DBMS who then handle database. A DBMS is made by a lot of components. First of all, there is the *optimizer* which receives in input the SQL instructions, transform the query in an internal representation and select the appropriate execution strategy for accessing data to answer queries. Then there is the *access method manager*, which performs the physical access to data implementing the strategy selected by the optimizer. After we find the *buffer manager*, which manage page transfer from disk to main memory and vice versa, it takes care of concurrent access to data (fundamental with writing operations) guaranteeing that different applications do not interfere among them. It guarantees also correctness of the database content when the system crashes. All this thing is done automatically by the database management system and this is the reason why we use this kind of software and we do not simply save data for example in one or many files.

DBMS works with transaction, which is a logical unit of work performed by an application. It is a sequence of one or many SQL instructions performing operations on the database. A transaction starts automatically with the first SQL instruction and ends when we perform "commit" or "rollback" operation. We do "commit" when everything is ok and we want to apply our SQL instruction on the database, otherwise we perform "rollback" when we do not want to apply changes on the database because something has gone wrong. Any transaction must guarantee what are called ACID properties: atomicity, consistency, isolation and durability. Atomicity means that a transaction cannot be divided in smaller units, it is not possible to leave the database in an intermediate state of execution. Consistency means that a transaction cannot violate integrity constraint on a database. This is ensured by defining integrity constraint in database schema. When a violation occurs, the system can *rollback* the transaction or try to automatically correct the violation. Isolation means that the execution of a transaction is independent of the concurrent execution of other transactions. Durability means that the effect of a committed transaction is not lost in presence of failures.

The DBMS chosen for this project is MySQL, perhaps the most popular SQL server, with InnoDB as engine. InnoDB creates automatically indexes for primary key, foreign key and unique constraint. Indexes are auxiliary structures that speed up searching operation on data. For this reason, we have an index on email attribute in user's table (because email is set to be unique in that table), while other indexes in other tables are or for primary key or for foreign key. We add by hand indexes on date because we perform queries on date when we run plugins.

5.3.2 Python connector and SQLAlchemy

To work with MySQL in Python we use both "Flask-SQLAlchemy" and "mysql-connector-python". The first one is a Flask extension that simplifies the use of SQLAlchemy inside the application. SQLAlchemy is a relational database framework that offers high-level ORM (Object Relational Mapper). ORM provides the data mapper pattern, where classes can be mapped to the database. With an ORM we can work with high-level SQL, we don't have to write by hand SQL query, but we handle Python object. We use this approach only for what concerns the user table, because, to handle user log-in and log-out, we use a very popular extension, called Flask-login, that needs object representing user to work properly. So, the most natural way to work was with SQLAlchemy. For all other stuff concerning weather forecast, we prefer to work directly with SQL language, to have a better control over data. In order to handle SQL in Python, we need a connector, a software layer that interface with MySQL server. Whit this connector it is possible to execute SQL command directly from Python application. We decide to use raw SQL language basically for one reason: the geographical data. We have to handle a lot of geographical information and we have to compute distance from all grid points (2850) to all weather stations (138), in order to fill the bridge table. Geographical information is saved in the database with “POINT” data type, provided by MySQL. To insert geographical data into table we have to use “`ST_GeomFromText(‘POINT(y x)’)`” function that construct geometry value starting from latitude and longitude. To compute distance between two geographical point, MySQL provides a dedicated function called “`ST_Distance_Sphere(point(long, lat), point(long, lat))`”. This function returns the distance in meters from two point. This function computes distance as the crow flies, it takes not into account road or traffic, but this don’t care for the purpose of this application. So, this function provide by MySQL is perfect in this use case.

5.4 Python multiprocessing

Compute the “bridge_ws_gp” table is a very expensive and time-consuming operation. To complete this task, we have to estimate the distance of each weather stations from each grid points using an SQL instruction: we have to perform 138 * 2850 = 393300 operations. If we do not adopt some strategies, this task can take up to 30 minutes in a medium-level personal computer[9].

To compute distance between points, we start from two Pandas data frame, one for the weather stations and another one for the grid points, both whose columns are id, longitude and latitude. The basic idea is to iterate over the weather station data frame and, for each row, iterate over the grid point data frame computing distance between points. The output is a new data frame which column are weather station’s index, grid point’s index and distance expressed in kilometres. Given this data frame, we sort rows for each different weather station’s index based on distance value. This approach has two big problems: the first one is that appending every time a new row in the result data frame is a very long operation, the second one is that this is clearly a candidate problem to be solved with parallelization.

[9] These results are not obtained doing professional test on different hardware and conditions, but are the simple evidence observed on the PC used to developed this application: CPU Intel i7-7500U 2.7GHz, RAM 8 GB, SSD 512 GB.

To solve the first problem, some “test” has been performed with different way to add new item to Pandas data frame. We found that the fastest way is to work with dictionaries and to build data frame from them. First thing to do is to declare an empty result data frame, which will be returned by the function and will contain the list of all weather stations with the nearest grid point of each and the relative distance. Then, we have to declare, for each new weather station, a list of dictionaries. After, we can estimate the distance between weather station and grid point and then, for every new single distance value computed, we create a new dictionary in which we add the weather station’s index, the grid point’s index and the distance value and then we append this new dictionary to the list of dictionaries declared before. Next, when we finish to iterate over grid points, we create a Pandas data frame starting from the list of dictionaries, we sort value based on distance, we keep only the first value (which represent the nearest grid point for the given weather station) and last we append this partial data frame to the final data frame that will be returned by the function. Working in this way speed up considerably the entire process, at least for what concerns saving partial results.

The second problem underlined before is completely different from the previous one. It is not specific about the Pandas libraries, but it is more general and quite diffuse in many programming areas. This task is clearly suitable to be execute in concurrent mode, because we can think to divide the weather stations data frame into a certain number of smaller data frame each one containing a subset of the whole weather stations. Then we can apply the function describe in previous paragraph on each one of these data frame to compute distance. This is exactly what we do in the application: before starting computing distance, we check how much weather stations we have to work with. The threshold is set to 30: if we have less than 30 weather stations, we do not use concurrent computation, otherwise we split the initial data frame into four data frames, each one with one quarter of the weather stations.

When we deal with concurrent computation, first of all we have to understand if our task is CPU-bound or I/O-bound. A task is CPU-bound if it spends most of the time doing computation using CPU, but it does a very little I/O operations, or maybe nothing at all.

On the other hand, a task in I/O bound if it does not perform operations that need a massive usage of CPU, but it is involved in I/O stuff, it waits for input and output operations to be completed. I/O functions include downloading data from network, transfer data from disk to main memory or vice versa, doing operations with file-system and so on. Multithreading can speed up programs that perform I/O-bound task, because with thread code is not really run concurrently, it is just an illusion: when one thread is waiting for an I/O operation to be completed, another thread is started. In fact, they are not executed at the same time, we can say that they just wait together and nothing more. We would not get as much speed up if we use multithreading to perform CPU-bound task, because all the threads run inside one single process. Whit multiprocessing, otherwise, we can use all the core presents in the CPU and we can effectively run code in parallel mode.

If we have to perform one task that simply wait for 1 second, Figure 11 shows what happens if we use multithreading, while Figure 12 shows the situation with multiprocessing.

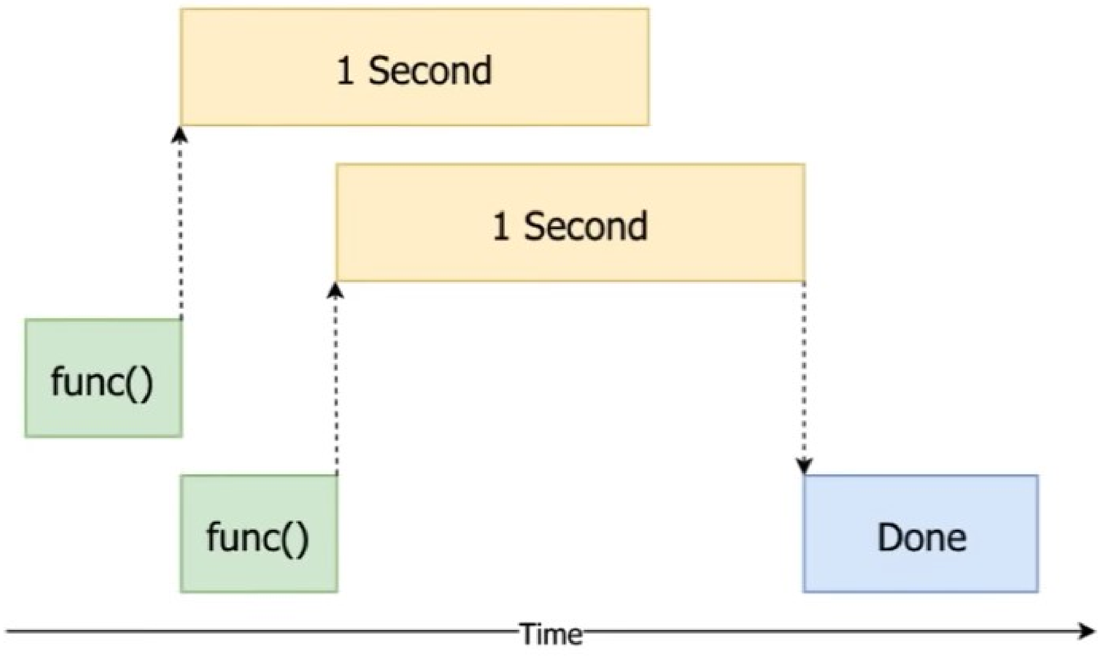


Figure 11: multithreading

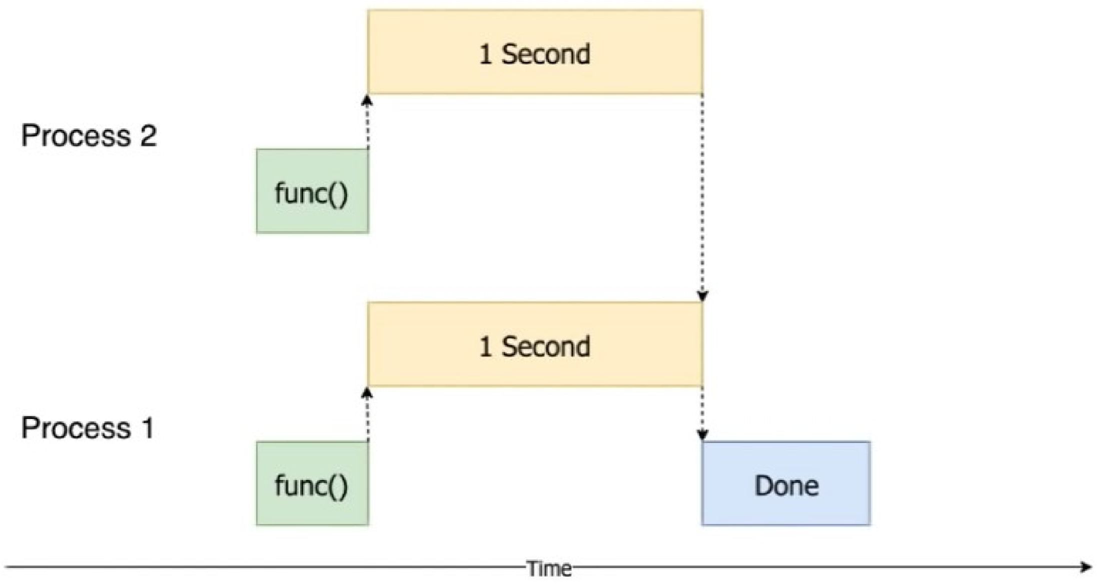


Figure 12: multiprocessing

Some benchmarks found on the web [6], even if they are quite old, show clearly the difference in using multithreading and multiprocessing when we deal with a CPU bound task. In this specific case, the author compares results of executing parallel a function that compute factorization of a number (this function is not optimized, but this do not care for the aim of this test). Results shown in Figure 13 are surprising. Splitting the computation to several threads slow down the performance, and the more threads are used, the slower the task runs. This is due to the way Python thread are implemented and the GIL (Global Interpreter Lock). It is a quite complex situation and this thesis does not want to go into it too much in detail. We just want to underline the final result. Otherwise, if we use multiprocessing, thigs are completely different. The more process we use, the more the application run faster. Obviously, there is a limit and we should find the right threshold. Result obviously are influenced by the specific task we want to perform and by the hardware we use, but also by the operative system, other programs in execution at the same time on the same PC and so on. There are many and many aspects to keep into account, and we must be aware of it. What we should keep from this article is the demonstration that Python threads are not good if we want to speed up our CPU-bound application (they actually slow it down), but we must use processes.

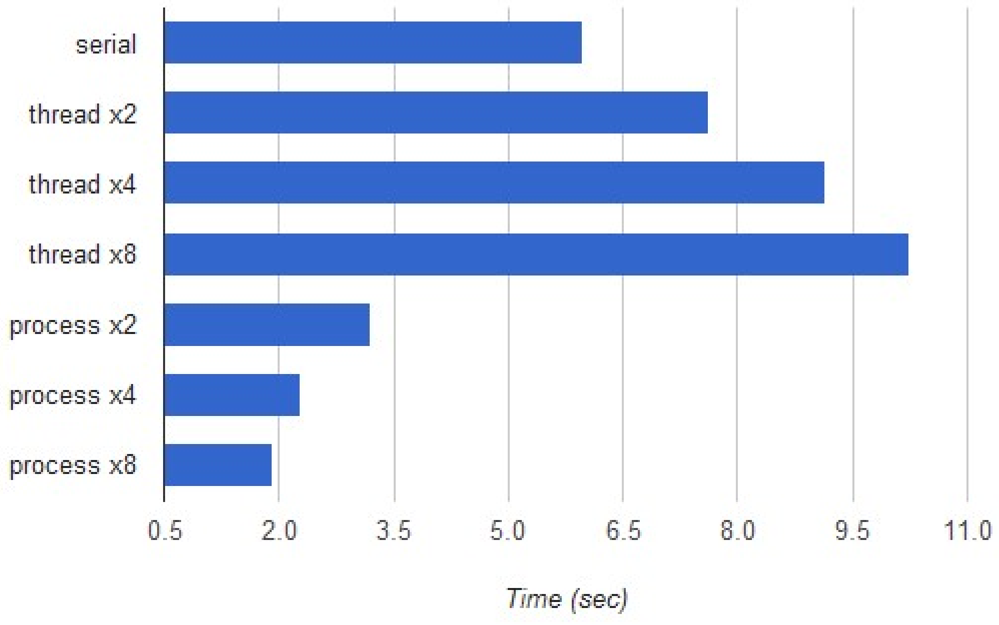


Figure 13: multithreading versus multiprocessing in a CPU bound task

Last thing we have to pay attention when we use multiprocessing computation is the way in which we can returning a result. If we create by hand a process using `multiprocessing` library, function we call cannot return a value with the `return` statement. The only way to get a result from the function is to pass a variable to every process in which they can store the result. But when we want to read that value, we must be sure that computation has been finished, otherwise we get an unpredictable result.

Starting from Python 3.2, the standard library introduced the future object in the modules `concurrent.future`. Future object is like a proxy for the result that is not known when we start a new process because the computation is not yet finished. Futures simplified the way in which we can get the result from a concurrent computation. Instead of creating processes calling the constructor of `multiprocessing` library, we can use the `ProcessPoolExecutor` from `concurrent.future`. In this way we can write the function to be executed concurrently like any other function, with the `return` statement at the end. Using ProcessPoolExecutor with the Python context manager, we can use the `submit` method to schedule a new function to be executed and to get the corresponding future object. On the future object we can then simply call the `result` method to obtain the value returned by the function.

Work with process with the `ProcessPoolExecutor` is simpler than creating it by hand. Future make code simpler and, consequently, reduce the possibility to make mistakes. For these reasons they should be used anytime we can.

5.5 Python scheduler

In this application, we have to regularly check if new weather forecast or measurement data are available, and if they are, we have to save them on the database. To perform this operation, we use the APScheduler (Advanced Python Scheduler) library [7], which can be installed throw the command "`pip install apscheduler`". This library allows to schedule Python code to be executed in a second moment, either just one time or periodically. This library has many types of different scheduler, some of them dedicated to specific framework like Qt. In our project we use the "BackgroundScheduler", which is not specific for any framework in particular and runs in background using a separate thread (when we call the `start` method of the scheduler, it will return immediately). The default executor of the background scheduler is the `ThreadPoolExecutor`, which uses futures like the processes explained before, but of course with thread and not with process. We basically need simply to check if there are new data and, if they are, we have to save them. We do a very little process over these data. So, we can say that operations performed by the scheduler are mostly I/O-bound and, for this reason, threads are the best choice.

In order to schedule a job, we have to choose a proper trigger for it. Trigger determines when the scheduled function has to be executed. APScheduler supports three type of triggers:

- date: useful when we want to execute the function just once in a specific moment

- interval: useful when we want to execute the function at fixed intervals of time
- cron: useful when we want to execute the function periodically, every day at a given time

In our application we scheduled two different functions, the first one that get new measured data from 3A society and the second one that check if new weather forecast data are available in the DET storage. For both we use, as said, the BackgrounScheduler with "cron" as trigger, because we want those functions to be executed every day at a given time. The first one starts at 7:00 AM, while the second starts at 2:00 AM. The first time has been chosen together with 3A society: it is good because at 7:15 AM they start the interpolation procedure and we do want to have modified data, but we prefer deal with raw information. The second time has been chosen because, at 2:00 AM most likely nobody is performing requests to the server, so we can spend part of CPU resource to upload new data. It is not rare to perform system update, uploading new data and other operations that need some resources when system is not overload.

5.6 Plugin architecture[10]

For the purpose of this applications, we need to plan the possibility to add new features to perform validation over collected data in a second moment. Plug-in architecture is what we were looking for, it is the best solution for this problem because gives the possibility to other people to write new Python files that can be easy add to our application and that can work together with it.

The basic idea under this solution is to have one folder in our application main directory that can collect all the new plugin organized in any way: they can be saved directly in the plugin folder or they can be put into an arbitrary number of subdirectories under the base folder. This because maybe one plugin can be quite complex and it can be composed of many files collecting together in a directory.

[10] The plugin solution has been found on GitHub https://github.com/gdiepen/python_plugin_example

The starting point for defining the plugin architecture is to create the base class `Plugin`, which contains the `title` and the `description` of the plugin itself, the `active` variable, which is a flag that is `True` if the plugin is active or `False` if is disable, the `filename` of the plugin python file and the `perform_operation` method, which implements the plugin logic. This basic class can be interpreted like a sort of Java interface, it is the starting point for all the plugin that will be added. Each one of them must inherit from this base class and must implement the `perform_operation` method, otherwise `NotImplementedError` exception will be raised.

To have a system that can dynamically load different modules at runtime, we have to introduce a new class, the `PluginCollection` class, that will take care of loading all plugin and enables them in order to apply the `perform_operation` method. The core of this class is the `walk_package` function, which look for all modules present in the plugin folder and, for each one, get all classes defined in the module and check if they are subclasses of `Plugin` class, but not the `Plugin` class itself, and if exist one *html* file in the *templates* directory which has the same name of the python file. Each class that satisfied these criteria is instantiated and added to a dictionary in which the key is just an id. In this way, plugin folder can contain also other classes that are not plugin (they can be support class for another plugin for example) and they will not influence the plugin framework.

Once that all plugin present directly in the package has been imported, application starts looking for the presence of sub-packages and inspects them by calling recursively the `walk_package` function.

The `PluginCollection` class is instantiate when the application starts in order to make all plugins available from the beginning.

All plugins are supposed to work with data of one single weather station, selected by user, over a period of time of one year, also decided by user from all the available years. All plugins get as input two Pandas dataframe, one for data coming from weather station and the other for weather forecast data.

The plugin execution work flows is the following:

- User selects one plugin among the active and the year on which he wants to execute it
- Application collects data related to that weather station for that year and calls the `perform_operation` method, giving to it the collected data
- Plugin performs all needed operations and must return a single result in the `result` variable
- Application performs the `render_template` method, calling the relative html file of the plugin giving to it the result

In this way application is capable to handle plugins without knowing what is the output of each single plugin and how it has to be displayed. It is up to the developer to implement the visualization of the obtained result.

5.7 Deployment

The developed application is cloud based and runs on a Linux server hosted by the Electronics and Telecommunications Department of Politecnico di Torino. We connect to this server through an SSH client. We installed the whole application by cloning it from the git repository we used for version control. After cloning the application, we create the Python virtual environment and installed all the dependencies in the virtual environment listed in the requirements file we made throw the command `pip freeze > requirements.txt`. In this way we obtain a text file containing all needed Python packages required by our application.

Even if Flask comes with an integrated web server useful during the development phase, this one is not adapted when we want to run our application on a production server because it was not built to be robust and performant enough. For this reason, we decide to substitute it with Gunicorn, which is a pure Python web server but it is robust and it is used by a lot of people as production server. Gunicorn can be installed with the command `pip install gunicorn`. We setup Gunicorn to listen requests in the internal network interface on port 8000. Gunicorn can be started by command line, but this is not the best option since we do not want to start it every time the server is rebooted. It would be better to have the server running in the background and have it under control in order to restart it if the system is rebooted. To do this, we provide a bash script which takes care of start Gunicorn and we schedule this script to be executed on system start-up. Gunicorn was developed to run behind a reverse proxy because it is designed for fast, low-latency client, but there also many clients that are neither fast or low-latency. For all these clients the reverse proxy is the solution. The developers themselves strongly recommend to use Nginx as reverse proxy within Gunicorn, because Nginx is capable of handling thousands of hundreds concurrent clients efficiently. To explain the importance of the reverse proxy, we can think to a slow client that interact whit the application: due to his slow connection, he keeps the Gunicorn worker busy for a long time and he make other client wait. The reverse proxy can buffer incoming requests from clients and the application responses and then he can send them at once, without making anyone wait.

Follow this suggestion, we adopt Nginx and we configure it to listen on port 80 and to redirect all clients' requests to the Gunicorn server. We do not use the https protocol on port 443 because we cannot buy an SSL certificate, which is needed to enable security protocol. To make Nginx serving a web site, we have to provide a configuration file in which we specify the TCP port (the 80 one in our case) and the actions to perform, which in our case are simply to forward all client requests to the Gunicorn server.

6 Conclusions

The developed application now is ready to start working: it is possible to perform any kind of validation over weather forecast by simply implement a Python script that performs operations over data and an html file to visualize results. Application is capable of handles all possible validations: there are no limits and no particular requirements, but the Python script which has to extends the base `Plugin` class and the html file.

With this effort we want to connect the meteorology world with the agrometeorology one in order to have a more performant and more powerful weather station, which keeps on collecting real-time data but which can also handle forecasting data thanks to the WRF model. If this works properly, there will be a lot of new possible improvements for what concerns the agrometeorological models. Until now, models worked only with real-time data to simulate phenological phases of plants and also of plant's disease, but if they can work with forecasting data, simulation can go further in the future, providing knowledges not available before. This could be of great help because it allows to planning in advance the correct crop management interventions depending on what will happen in the future. It is reasonable to think that acting in advance would be better, because it allows to prepare farmers adequately and better for what will happen. Therefore, before using weather forecast information to run models, it is essential to know if weather forecasts are good or not and how much they are reliable, because they are very influenced by the geographical conditions: weather forecast computed on a flat territory could perform very differently from weather forecast computed on a mountainous territory. Another very interesting aspect to examine is the presence of regular error pattern: what we expect while performing validation is to find some behaviour of weather forecast that are always the same, for example an overestimation of rain or temperature. If validation is capable to find these patterns, it will be easy to take action to correct them obtaining in this way a good and reliable value. Or maybe it will be found that not all forecasting variables are reliable in the same way, because the WRF model can easily predict some kind of variable better than other or because not all forecasting variables have the same behaviour in the geographical zone in which they are computed. There are many aspects that need to be considered.

These are just few examples to underling how indispensable is the validation before using operatively weather forecast information.

With this application we realized the starting point for trying to connect together meteorology and agrometeorology, with the aim to obtain a more performant weather station that could help more and more the crop grow. We solved all problems related to acquiring data from weather stations and from WRF model, to interpreting data, to parsing them and to store them in a properly well-organized database. We solved problem of finding the nearest grid point for each station and we made possible to change this association every time user wants. We provided also a user-friendly interface to operate with and we made simple to introduce new plugins that can extend application adding new functions. This software architecture is perfect to do what this application is supposed to: allow weather forecast validation in an easy way in order to use them in future for support agrometeorological mathematical models.

7 Bibliography

[1] J. G. Powers, “The weather research and forecasting model. Overview, system efforts and future directions,” *American meteorological society,* pp. 1717-1737, 2017.

[2] “Weather research and forecasting model,” [Online]. Available: https://www.mmm.ucar.edu/weather-research-and-forecasting-model. [Accessed 2020].

[3] “Network Common Data Form (NetCDF),” [Online]. Available: https://www.unidata.ucar.edu/software/netcdf/. [Accessed 2020].

[4] “La rete agrometeorologica del Piemonte (RAM),” [Online]. Available: https://www.regione.piemonte.it/web/temi/agricoltura/agroambiente-meteo-suoli/rete-agrometeorologica-piemonte-ram. [Accessed 2020].

[5] “3a,” [Online]. Available: https://www.green-planet.it. [Accessed 2020].

[6] “Python - parallelizing CPU-bound tasks with multiprocessing,” [Online]. Available: https://eli.thegreenplace.net/2012/01/16/python-parallelizing-cpu-bound-tasks-with-multiprocessing/. [Accessed 2020].

[7] “Advanced Python Scheduler,” [Online]. Available: https://apscheduler.readthedocs.io/en/stable/index.html. [Accessed 2020].

[8] S. Orlandini, L. Massetti and A. Dalla Marta, “An agrometeorological approach for the simulation of Plasmopara viticola,” *Computers and electronics in agriculture,* pp. 149- 161, 2008.

[9] V. Rossi, T. Caffi, S. Giosuè and R. Bugiani, “A mechanicistic model simulating primary infections of downy mildew in grapevine,” *Ecological Modelling,* pp. 480-491, 2008.