



## Deliverable D8.8 – WP8

# REPORT ON THE COMPLETE VERSION OF THE AGRO-ECOLOGICAL MODELLING PLATFORM API SOLUTION, AS THE FIRST COMPONENT OF THE CCP VRE, AND THE ASSOCIATED USER GUIDE (IRET-CNR-FI).

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System - CUP B53C22002150006 (D.D. n. 130/2022)  
Funded by EU - Next Generation EU  
Mission 4 “Education and Research” - Component 2: “From research to business” -  
Investment 3.1: “Fund for the realization of an integrated system of research and innovation infrastructures”



<b>Deliverable number:</b>	D8.8
<b>Work package:</b>	WP8
<b>Intermediate Objective:</b>	IO8.6
<b>Deliverable type:</b>	X Document, report
	Websites, patent filings, videos, etc.
	Other: please specify .....
<b>Dissemination level:</b>	X Public
	Restricted
<b>Estimated delivery (bimester):</b>	B9
<b>Actual delivery date:</b>	30.04.2024
<b>Author(s) (Partner-OU):</b>	<u>ALESSANDRO MONTAGHI</u> CONTRIBUTING AUTHORS: ELENA PAOLETTI, MARCELLO DONATELLI, DARIO DE NARD, DAVIDE <u>FANCHINI</u>
<b>Reviewed by:</b>	ITINERIS EXECUTIVE BOARD
<b>Note:</b>	

## Contents

1. <i>PROGRESSES OF THE ITINERIS PROJECT – WP 8, TASK 8.3</i> .....	4
<b>Task 8.3 Overview of the work package</b> .....	4
2. <i>DESCRIPTION OF THE PLATFORM CREATED IN THE FRAMEWORK OF THE ITINERIS PROJECT – WP 8 AND TASK 8.3</i> .....	6
<b>Agro-Ecological Modelling platform Components</b> .....	6
<b>Agro-Ecological Modelling platform APIs endpoint</b> .....	12
3. <i>USER GUIDE</i> .....	13
<b>Data Access API</b> .....	13
<b>Infections Models</b> .....	19
<b>Weather Anomalies Checks and Corrections</b> .....	27
<b>Image classification services</b> .....	36

## 1. PROGRESSES OF THE ITINERIS PROJECT – WP 8, TASK 8.3

### Task 8.3 Overview of the work package

The Work Package 8.3 - Crop, Plants and Pests VRE (CPP VRE) is part of WP 8 (Virtual Research Environments and Cross-disciplinary Activities) of Italian Integrated Environmental Research Infrastructures System (ITINERIS). ITINERIS is a project funded by EU - Next Generation EU PNRR- Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realization of an integrated system of research and innovation infrastructures”.

The main goal of WP 8.3 is to make accessible an array of cross-platform modelling solutions, data transformation tools, and cloud hosted computational facilities to contribute to crop production, plant phenology, pest and disease spread, and cropping system. Specifically, crop production, plant phenology, pest and disease spread, and cropping system management is at the heart of crucial scientific and applied challenges, also considering the European “Farm-to-Fork” Strategy and the activities of the Agritech National Centre.

The CPP VRE will contribute to the system by making accessible an array of cross-platform distributed modelling solutions, data transformation tools, and cloud hosted computational facilities, allowing users of the cross-RI integrated VRE system to perform scenario analysis and digital twin components in an interactive way. Proposed modelling solutions will include well established process-based models for crop production, water use, plant phenology, pest and disease spread, pathogens dynamics and impact, and cropping system management as well as advanced statistical methods for image classification and time series analysis, and they will be published with the SaaS (Software as a service) paradigm, enabling researchers to integrate them into different technological stacks, including VREs.

Such models are built in the BioMA (Biophysical Model Applications) framework, which is currently adopted by the JRC of the European Commission and widely regarded as a gold standard in agroecological modelling. Models in BioMA are composed of small components, which could be recombined (modularity) to increase reusability and sharing. Bioma's implementation features enabled it to be extended and adapted to meet the CPP VRE requirements. This activity will include two main lines of action: modelling solution development and publication, and data tools development and publications; both activities will build on top of existing internal tools and know-how and will represent the largest

knowledge and expertise transfer activity ever undertaken within AnaEE (Analysis and Experimentation on Ecosystems).

The CPP VRE will also make available a second set of cloud services meant for data access, retrieval, and quality control, that following the SaaS (Software as a Service) paradigm will be actionable also from any other VRE. Meteorological and satellite data and related utilities for data processing will be made available, allowing researchers from different domains and RIs as well to access curated data transformation pipelines and fostering the development of Big Science practices in the environmental domain. All data resources comply to FAIR principles as developed in ENVRI-FAIR project of the EU.

The CPP VRE was realized through the implementation of the Agro-Ecological Modelling platform API. This platform is based on the expertise, both technical and scientific, previously acquired for the creation of BioMA and for the management and implementation of the European research infrastructure AnaEE.

In this regard, BioMA is a public domain software framework designed and implemented for developing, parameterizing, and running modelling solutions based on biophysical models in the domains of agriculture and environment. It is based on discrete conceptual units codified in freely extensible software components. The goal of this framework is to rapidly bridge from prototypes to operational applications, enabling running and comparing different modelling solutions. A key aspect of the framework is the transparency which allows for quality evaluation of outputs in the various steps of the modelling workflow. The framework is based on framework-independent components, both for the modelling solutions and the graphical user's interfaces. The goal is not only to provide a framework for model development and operational use but also, and of no lesser importance, to provide a loose collection of objects re-usable either standalone or in different frameworks. The software is developed using Microsoft C# language in the .NET framework.

Moreover, AnaEE (Analysis and Experimentation on Ecosystems) is a European research infrastructure on terrestrial and aquatic ecosystems that integrates national experimental (open for in situ and closed in vitro ecosystem studies), analytical and modeling platforms. It aims to provide support to the scientific community in using these platforms by offering services developed by three supranational Centers, Technology Centre, Data and Modeling Centre, and Interface and Synthesis Centre, under the coordination of the Central Hub. Specifically, we have that the Central Hub (CH) is at the heart of the strategy, coordination, communication, and administration of AnaEE-ERIC. It also manages the AnaEE web portal that gives access to all the resources of the organization. With a distributed infrastructure including over 100 national

facilities, 3 supranational Service Centers and a variety of users and stakeholders, the development and operation of this Central Hub is crucial for the overall success of AnaEE. the role of the AnaEE Technology Centre (TC) is to watch and develop new emerging technologies and ensure that instrumentation and methods are coordinated among the facilities. The AnaEE-TC is also responsible for the spin-off of new technologies developed within AnaEE, as well as for coordinating the training of users and facility operators. The AnaEE Interface and Synthesis Centre (ISC) is responsible for the overall integration of the results obtained thanks to AnaEE RI. It prepares synthesis and opinion papers on behalf of AnaEE, watches for emerging societal needs, answers to demands from the society, economy, and policy makers. It is also responsible for the training and outreach. The AnaEE Data and Modelling Centre (DMC) is responsible for the processing of the data and metadata, the provision of data to the users (either the direct users or the community), the access to the models and model factory. It also organizes workshops and training for users and AnaEE staff.

## 2. DESCRIPTION OF THE PLATFORM CREATED IN THE FRAMEWORK OF THE ITINERIS PROJECT – WP 8 AND TASK 8.3

### Agro-Ecological Modelling platform Components

According to the technical requirements defined by the project, the Agro-Ecological Modelling platform was implemented to take into account the principles of Findability, Accessibility, Interoperability, and Reusability (FAIR).

The FAIR data principles, defined in 2016 by a consortium of scientists and organizations, describe distinct considerations for contemporary data publishing environments with respect to supporting both manual and automated deposition, exploration, sharing, and reuse. In detail, according to the FAIR guiding principles, data need to be (1) Findable (e.g., data are described with rich metadata), (2) Accessible (e.g., data and metadata are retrievable by their identifier using a standardized communications protocol, and the protocol is open, free, and universally implementable), (3) Interoperable (e.g., data and metadata use a formal, accessible, shared, and broadly applicable language for knowledge representation), and (4) Reusable (e.g., data and metadata are released with a clear and accessible data usage license). Note that these high-level FAIR Guiding

Principles precede implementation choices, and do not suggest any specific technology, standard, or implementation-solution. Note that these high-level FAIR Guiding Principles precede implementation choices, and do not suggest any specific technology, standard, or implementation-solution. However, these principles can be used as a guide for the design and development of the proposed platform architecture.

Agroecological modeling API platform is based on the following technological solutions adopted in the implementation phase. Detailed information on the technology solutions adopted and the reasoning behind their choices is provided in the following section.

- 1) **APIs:** API stands for Application Programming Interface. In the context of APIs, the word Application refers to any software with a distinct function. Interface can be thought of as a contract of service between two applications. This contract defines how the two communicate with each other using requests and responses. API architecture is usually explained in terms of client and server. The application sending the request is called the client (i.e., a program that exchanges data with a server through an API), and the application sending the response is called the server i.e., the computer that runs an API). So, in the WP 8.3 example, the Agro-Ecological Modelling platform API is the server, and the researcher application is the client. In details, APIs are mechanisms that enable two software components to communicate with each other using a set of definitions and protocols, where a computer protocol is an accepted set of rules that govern how two computers can interact with each other.

On the Internet, the main protocol is the Hyper-Text Transfer Protocol better known by its acronym, HTTP. Communication in HTTP centers around a concept called the Request-Response Cycle. The client sends the server a request to do one or more actions. In response, the server lets the client know whether or not the server can accomplish the request. The request consists of a URL (i.e., uniform resource locator is a unique identifier used to locate a resource on the Internet), a request method (i.e., the request method tells the server what kind of action the client wants the server to take), a list of headers (i.e., headers provide meta-information about a request. They are a simple list of items like the time the client sent the request and the size of the request body), and a body (i.e., the data the client wants to send the server). HTTP responses have a very similar structure to requests. The main difference is that instead of a method and a URL, the response includes a status code. Status codes are three-digit

numbers that each have a unique meaning. For example, when a status code of 200 (i.e., OK) is returned, that means the request was successfully processed. Conversely, when a status code of 404 (i.e., not found) appears, that means the requested resource was not located (this means that the request the client sent was received by the server, but it could not find the service), and 400 (i.e., bad request) if an unsupported content type is requested or for any other invalid request.

The concepts described above are summarized in the term HTTP-based APIs. There are four different ways that HTTP-based APIs can work which are SOAP APIs, RPC APIs, WebSocket APIs, and RESTful APIs. Among these solutions, the RESTful APIs are the most popular and flexible APIs used in real-world applications. The client sends requests to the server as data. The server uses this client input to start internal functions and returns output data back to the client.

- 2) RESTful APIs: a RESTful API, also known as a REST API, is an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services. REST stands for REpresentational State Transfer and was created in 2000 by computer scientist Roy Fielding in his PhD dissertation. While REST is not restricted to HTTP-based APIs, they are both closely linked in practice. RESTful APIs now drive many web servers on the internet, and they are an industry standard that can be easily integrated into virtually any software product and in scientific platforms. The ideas in REST aim to be simple and to decouple the API from the underlying services that serve the API. It uses a request-response model and is stateless, as all the information necessary to do something is contained within the request.

REST architectures are based on five principles: (1) everything is a resource, (2) each resource is identifiable by a unique identifier, referred to as URI (Uniform Resource Identifier), (3) use the standard HTTP methods, (4) resources can have multiple representations, and (5) communicate stateless. In Principle 1, the key abstraction of information in REST is a resource. Usually, a resource is something that can be stored on a computer and represented as a stream of bits: a document, a row in a database, or the result of running an algorithm. For example, in the Agro-Ecological Modelling platform API a resource can be the result of processing an infection model. Following the Principle 2, each resource, to be a resource,

it must have at least one URI, where the URI is the name and address of a resource. For Principle 3, the client (e.g., a researcher) can use different HTTP methods to manipulate the resource identified by a URI. The most commonly used HTTP methods in REST include: GET (used to retrieve a representation of the resource, and when performing a GET request, server should respond with the result in the form of JSON), POST (used to create a new resource), PUT (used to updates a resource or creates it as an identifier provided from the client), and DELETE (used to delete a resource). The APIs of the Agro-Ecological Modeling platform only exposes GET and POST endpoints. The following table illustrates what has been described:

<i>HTTP verb</i>	<i>Action</i>	<i>Response status code</i>
GET	Retrieve information about the REST API resource	200 (OK) if the resource exists, 404 (Not Found) if it does not exist, and 500 (Internal Server Error) for other errors.
POST	Create a REST API resource	201 (Created) if a new resource is created, 200 (OK), if updated, and 500 (Internal Server Error) for other errors.
PUT	Update a REST API resource	201 (Created) if a new resource is created, 200 (OK) if the resource has been updated successfully, 404 (Not Found) if the resource to be updated does not exist, and 500 (Internal Server Error) for other errors.
DELETE	Delete a REST API resource or related component	200 (OK) or 204 (No Content) if the resource has been deleted successfully, 404 (Not Found) if the resource to be deleted does not exist, and 500 (Internal Server Error) for other errors

In Principle 4, resources can have multiple representations such as JSON (JavaScript Object Notation), CSV (comma-separated values), XML (eXtensible Markup Language), and HTML (HyperText Markup Language), among others. The choice of representation format depends on the specific requirements of the client and server. Agro-Ecological

Modelling platform API primarily uses the JSON format to represent resources. Finally, in Principle 5 the communicate request-response model is stateless, as all the information necessary to do something is contained within the request.

The choice to implement the Agro-Ecological Modelling platform according to Restful API principles was based on the following advantages: simplicity, scalability, flexibility, technology independence, visibility, ease of integration, and well-supported. Indeed, RESTful APIs use standard HTTP methods and URIs which in turn makes the client server-interactions very simple. The Stateless nature and resource-based architecture allow for horizontal scaling, making RESTful services suitable for large-scale applications. RESTful services can be easily scaled horizontally, as they are stateless. Each request from a client contains all the information needed to fulfill that request, which makes it easier to distribute and load balance. REST can be used with various data formats, such as JSON, XML, and HTML. Clients and servers can communicate using different representations of the same resources. Moreover, the REST architectural style is technology independence since it can be used with any programming language and technology stack that supports HTTP. Additionally, visibility of the service means that every aspect of it should self-descriptive and follow the natural HTTP language. Finally, REST is widely supported by most of the web development frameworks and libraries.

From the description of the advantages presented by the RESTful API-based implementation, this solution is best suited to integrate with VRE of ITINERIS.

- 3) Domain model: the domain model of the Agro-Ecological Modelling platform is a cloud extension of BioMA, specially made for the ITINERIS project. BioMA is a modelling framework composed of models and applications designed for running, calibrating, and improving biophysical and crop growth models. Biophysical models are algorithms to simulate a part of the biophysical system. Such algorithms can be coded into discrete software components. The BioMA modelling framework is based on independent components, for both modelling solutions and the graphical user interface. The framework also includes integrated tools to support the calibration of the models, to run the simulations at grid level, and to

visualize the results. All services of the extended version of BioMA for Agro-Ecological Modelling platform are exposed by RESTful API.

- 4) **Data representation:** the most common data formats in modern APIs are JSON (JavaScript Object Notation) and XML (Extensible Markup Language). However, JSON format has become the de facto standard for API responses, due to its lightweight nature and easy readability. It serves as a universal language for data interchange between servers and web applications. JSON is a standard text-based format for representing structured data based on JavaScript object syntax. It is commonly used for transmitting data in web applications. The main advantages of having implemented the platform using a data representation in JSON format are the following: human-readable (JSON is self-describing and easy to understand, even for those who are not developers), lightweight (its simplicity allows for quick parsing and a smaller data footprint compared to other formats like XML), and language-agnostic (JSON is supported by most programming languages, making it highly versatile for backend and frontend development). When an API is called, the Agro-Ecological Modelling platform responds with a JSON-formatted text that represents the data requested.
  
- 5) **Cloud API management:** API Management is a solution encompassing the collections of tools used to design and manage APIs, referring to both the standards and the tools used to implement API architecture. Client requests are handled by the API management, which forwards them to the domain model, which processes them and returns them to API management. API management then returns the response to the client in the most correct form. The API management implemented for the realization of the platform considered the following components: API design, API gateway, API analytics, and API catalog. API design includes the ability and features like importing an API from specifications, create API endpoints, define service contracts, and generate documentation. An API gateway allows configuring an API gateway engine, manipulating requests and responses, URL rewriting, caching, security enforcement and pre-authentication, and applying request-based security rules. API analytics includes the analytics of API usage, which is often configured as part of the API gateway and tracked and monitored. Analytics provide usage and telemetry insights and reporting dashboards.

## Agro-Ecological Modelling platform APIs endpoint

The endpoints developed for the Agro-Ecological Modelling platform APIs are as follows:

1. **Infections Models:** this endpoint incorporates different models of phytopathology infection in agricultural settings. Agro-Ecological Modelling platform hosts a growing collection of plant infection models (e.g., Oidio, Peronospora, and so on) tuned to perform well over the Italian territory.
2. **Advanced Infections Models:** this endpoint includes a new set of infection models including crop phenology.
3. **Vine varieties classifier:** a computer vision model to identify grapevine variety from leaf images. Leaf images must be full frontal and in good light conditions. It identifies 27 distinct varieties from the CREA (Council for Agricultural Research and Economics) collection.
4. **Weather Anomalies Check and Corrections:** service to check anomalies in a weather data series and correct them according to a set of rules.
5. **Cloud Service Optimizer:** service to run optimization jobs over APIs exposed through the platform. Its main purpose is to allow model calibration.
6. **Model APOs-v1:** APIs to configure and run model executions.

<i>Name</i>	<i>Short description</i>	<i>Type</i>
Infections Models	set of infection models	REST
Advanced Infection Models	new set of infection models with crop phenology	REST
Vine varieties classifier	grapevine varieties identification from leaf	REST
Weather Anomalies Checks and Corrections	check and correct data weather anomalies	REST
Cloud Service Optimizer	model optimizer	REST
Model APIs - v1		REST

### 3. USER GUIDE

#### Data Access API

In this notebook we show how to call the weather data quality check and correction API in a Python environment. The configuration (detectors and correctors) and the weather series to be corrected are already prepared in the folder "Files" with the filename "WeatherQuality\_configuration.json". This configuration will be used to prepare the Client with all the information to be passed to the service.

#### Setting up the environment

1. Install dependencies.
2. Create Environmental Variables.

These operations must be done once: after you install dependencies and declare environment variables, they will be there forever, and you will only have to import them to allow your code to use them.

#### Installing dependencies

In what follows, the exclamation mark ! is used to send command directly to the Operating System. The following command installs the required dependencies detailed in the requirements.txt file present in the project root folder. Put the cursor in the following cell and press the Run button:

```
! pip install -r requirements.txt
```

Right now, you should have all the required dependencies installed on your interpreter, but they are not yet visible, i.e., your code does not know where to find them. To be able to use them, you need to import them as follows:

```
import os
import re
import requests
from io import BytesIO, StringIO
import pandas as pd
import json
import shapely
import matplotlib.pyplot as plt
import shapely.wkt
```

If the above cell raised an error, it means that something went wrong while installing the dependencies.

### Setting up the API personal key

This is the key provided by the Developer Portal when the relative subscription is activated. It must be included in every request sent to the APIs.

The best way to use personal keys is to store them as environment variables in the Operative System, so all programs installed should be able to read them, with no need of explicitly writing them in the code or configuration files. There are different ways to set an environment variable:

1. Windows: go to Properties -> System and look for Advanced System Settings in Windows, at the bottom of the dialog there is a button named Environmental Variables, click it and add the required variable. Alternatively, this can be done by opening a cmd prompt and type `set NAME "VALUE"` and the variable will be visible to all subsequent sessions (i.e. you should close that command prompt).
2. Linux: run in your shell `export NAME=VALUE`

If running under Windows, replace "your api key" with the correct value and run the following cell:

```
!setx AGRIDIGIT_API_KEY "your-api-key"
```

The jupyter server must now be restarted (also the cmd), to make the new environment variable visible.

If are running under LINUX, replace your api key with the correct value and run the following cell:

```
!export AGRIDIGIT_API_KEY=your-api-key
```

Now the relevant variables can be made available to the code in this example:

```
SERVICE_KEY = os.environ['AGRIDIGIT_API_KEY']
```

## Calling the API

We are going to use RESTful Web Services, i.e. machine-operable services that are on the internet and work with http protocol that is the same used by Web browsers.

```
SERVICE_URL = 'https://api.progettoagridigit.it/crea-aa-dailyweather'
```

## Writing a remote service access method

Consuming a Web Service requires a little coding effort that we can wrap up in a function to call anytime we need to ask something to the API. This will make the code way easier to maintain and to read. Following are technical details on such a function.

Defining a function in Python is straightforward: start with the keyword `def`, than enter the name of the function, and its arguments between brackets. Finally put a colon at the end of the argument declaration. We are defining the function with several arguments with default values. Python allows default values for function arguments.

RESTful Web services are stateless services that are exposed on the Web and actionable with an http request, i.e. the same technology Web browsers use to request web pages and applications. The stateless bit is referred to the communication protocol between the client and the server (in this case the API) and it means that the client-server communication is constrained by no client context being stored on the server between requests. The practical implication is that if two distinct clients make the very same request at a certain time they will receive the same response. On the other hand the server can be stateful, for instance it may have a database and present users with different results over time as its data is updated.

RESTful Web Services use JSON as data exchange format: they expect the Client to send requests in JSON form, and in return they give their response in JSON formats. In Python JSON objects are mapped into Dictionaries, which are a primitive type. Dictionaries are made like this:

```
{  
    key_1: value_1,  
    key_2: value_2,  
    ...  
    key:n : value_n  
}
```

Where keys and values can be objects of any type, with only two major restrictions:

1. A given key can appear in a dictionary only once. Duplicate keys are not allowed. A dictionary maps each key to a corresponding value, so it doesn't make sense to map a particular key more than once. If a key is specified a second time during the initial creation of a dictionary, then the second occurrence will override the first.
2. A dictionary key must be of a type that is immutable. For example, we can use an integer, float, string, datetime, or Boolean as a dictionary key. However, neither a list nor another dictionary can serve as a dictionary key because lists and dictionaries are mutable. Values, on the other hand, can be any type and can be used more than once.

Dictionaries can be naturally converted to JSON objects and vice versa, and we will leverage this feature to build our request function.

Python comes with a library named requests that allows to make http requests with ease. We will need to:

1. set authentication properties, i.e. the API key, in the request header.
2. "dump" the request dictionary into a JSON object.
3. Invoke the Web service and read its status.

4. If the request has been successful, we can parse the JSON response into a dictionary and return it. Some requests may have a non-JSON response. In such a case, the response will be converted into a string. If it is impossible to convert the response into a string, we will print a warning and return a None object.

If the request failed, we will print a warning and return and return a None object.

The following function implements these steps.

```
def call_restful_api(function, data={}, base_url=SERVICE_URL, key=SERVICE_KEY, method='POST'):
    ...

    Calls a RESTful API
    Parameters:
        function (string): the name of the API method to invoke
        data (dictionary): the data payload
        base_url (string): the base address of the API
        key (string): the service key you registered
        method(string): the HTTP method you want to use, defaults to POST

    Returns:
        service response (str or dictionary): Web service response, either a dictionary or a string.

    ...

    # set request headers
    headers = {'Content-Type': 'application/json'}
    headers['Ocp-Apim-Subscription-Key'] = key
    # serialize request data into a json object
    payload = json.dumps(data)
    response = None
    # invoke the Web Service
    if method == 'POST':
        response = requests.post(base_url + '/' + function, data=payload, headers=headers)
    else:
        response=requests.get(base_url + '/' + function, data=payload, headers=headers)
    if response.status_code == 200: # the 200 status means 'success'
        try:
            return response.json()
        except:
            try:
                return response.text
            except:
                print('Unable to decode server response')
    else:
        print('Request failed with code ' + str(response.status_code))
    return None
```

The call to the Web Service is done by the following line:

```
response = requests.post(base_url + '/' + function, data=payload,  
headers=headers)
```

Where payload is the request JSON object, and headers is a dictionary of headers that, in our case, includes the content type of the body and the authentication key. Such a call returns a Response object that has three objects of interest:

1. The Response code: an HTTP code, 200 means "ok", 404 is "not found, check the URL", 403 means "wrong key, probably you are not worthy of using the system", 401 means "invalid or utterly moronic request", and 50X that something very bad occurred server side.
2. The elapsed time.
3. The response object, which can be interpreted as a string or as a dictionary.

Depending on the response code and response object type, the above function returns either a string, a dictionary or a None object.

## Infections Models

In this example we will showcase how to call the API Infections Models. Agro-Ecological Modelling platform hosts a growing collection of plant infection models tuned to perform well over the Italian territory, such models are used by our personnel to build service like DSS, which runs the infection models over the whole Italian territory. In this example we'll show how to run a model on a specific site.

We assume that you have already set up properly your environment, including both dependencies and global variables, if you have not done it yet, check out the first notebook in this series and follow the procedure therein described.

The following code snippet imports all the required dependencies.

```
# statistical processing components
import numpy as np
import pandas as pd

# presentation components
import matplotlib as mpl
import matplotlib.pyplot as plt
# this option allows plots to be embedded in the notebook file
%matplotlib inline

# http request related components
import requests
import json

# regular expressions
import re

#date and time
import datetime

#input and output low level util
from io import StringIO

#OS utilities. We'll use it just to retrieve env variables.
import os
```

Now it is time to read from the environment variables the API key.

```
# important variables
SERVICE_URL = 'https://api.progettoagridigit.it/infectionsmodels'
SERVICE_KEY = os.environ['AGRIDIGIT_API_KEY']

def call_restful_api(function, data={}, base_url = SERVICE_URL, key=SERVICE_KEY, method='POST'):
    # set request headers
    headers = {'Content-Type': 'application/json'}
    headers['Ocp-Apim-Subscription-Key'] = key
    # serialize request data into a json object
    test_data = json.dumps(data)
    response = None
    # invoke the Web Service
    if method == 'POST':
        response = requests.post(base_url + '/' + function, data=test_data, headers=headers)
    else:
        response=requests.get(base_url + '/' + function, data=test_data, headers=headers)
    if response.status_code == 200: # the 200 status means 'success'
        try:
            return response.json()
        except:
            try:
                return response.text
            except:
                print('Unable to decode server response')
    else:
        print('Request failed with code ' + str(response.status_code))
    return None
```

## Data preparation

All Infection Models expect their input data to be passed as tables, more specifically, these models expect:

1. A weather table containing weather data.
2. A locations table containing information about the location the weather data refers to.

This information must be packaged in a dictionary that should look like this:

```
{'Tables' :[
  {
    'ColumnNames': [list of weather variables labels],
    'Rows': [
      [values for row 1],
      [values for row 2],
      ...
      [values for row n]
    ],
    'Name': 'weather'
  },
  {
    'ColumnNames': [list of locations variables labels],
    'Rows': [
      [values for row]
    ],
    'Name': 'locations'
  }
]
```

Mind that the locations table should include only one row, as each infection model call is supposed to evaluate infection risk over a single location.

```
data = {"startTime":"2022-02-14",
"endTime":"2022-07-21",
"timeStep":"hour",
"wkt":"POINT(9.276383054989505 44.39341184704442)",
"limit":1,
"mode":"bioma" # this is the output format
}
# call the service
model_data = call_restful_api(
  'getNearestStationData',
  data,key = SERVICE_KEY,
  base_url = 'https://api.progettoagridigit.it/crea-aa-dailyweather',
  method= 'POST'
)
```

The previous format can be easily converted into pandas data frame object and plotted to better visualize the data we are going to pass to the model.

```

data = {"startTime":"2022-02-14",
"endTime":"2022-07-21",
"timeStep":"hour",
"wkt":"POINT(9.276383054989505 44.39341184704442)",
"limit":1,
"mode":"bioma" # this is the output format
}
# call the service
model_data = call_restful_api(
    'getNearestStationData',
    data,key = SERVICE_KEY,
    base_url = 'https://api.progettoagridigit.it/crea-aa-dailyweather',
    method= 'POST'
)

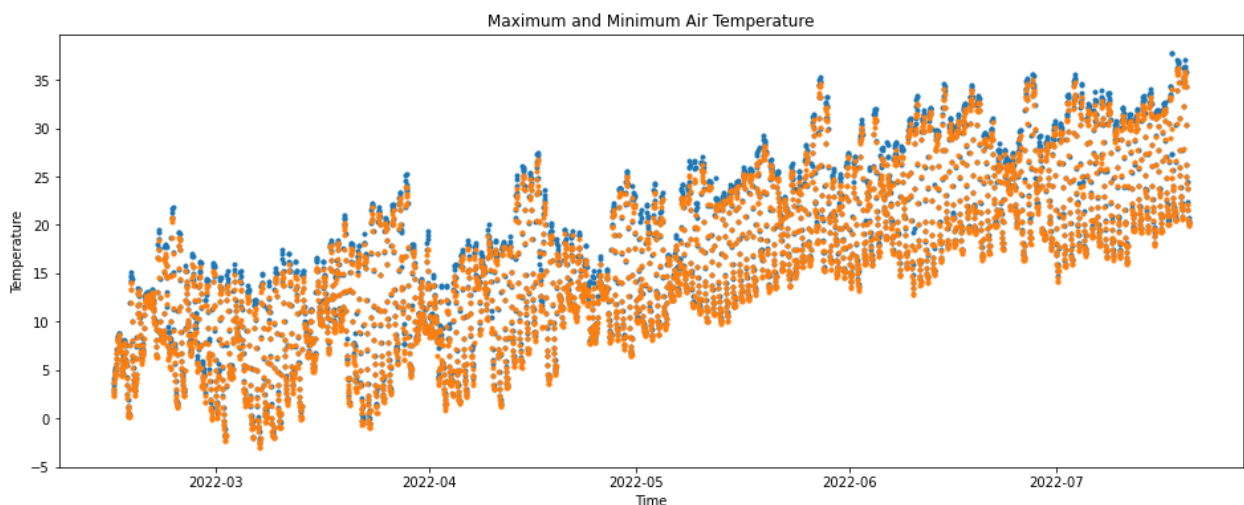
```

```

# extract the actual table containing weather data
weather_table = list(filter(lambda table: table['Name'] == 'weather', model_data['Tables']))[0]
# telling pandas which are the fields containing column names and actual rows
weather_data_pd = pd.DataFrame(weather_table['Rows'], columns = weather_table['ColumnNames'])
# parsing dates...
weather_data_pd['Date'] = pd.to_datetime(weather_data_pd['Date'])
# ... and setting them as rows index
weather_data_pd = weather_data_pd.set_index('Date')

plt.figure(figsize = (16, 6))
plt.title("Maximum and Minimum Air Temperature")
plt.plot(weather_data_pd.index, weather_data_pd['AirTemperatureMaximum'], marker='.', linestyle='None')
plt.plot(weather_data_pd.index, weather_data_pd['AirTemperatureMinimum'], marker='.', linestyle='None')
plt.ylabel('Temperature')
plt.xlabel('Time')
plt.show()

```



## Downey Mildew model

Now we can feed the Peronospora model with the data we received from the weather Web service.

```
response = call_restful_api(
    'Peronospora',
    data = model_data,
    method='POST'
)
```

This model evaluates the chances of Downey Mildew outbreak, signals when the fungus is likely to show up, and counts the number of outbreak events within the provided time frame. This information are provided as a table, which, as you may imagine, can be easily converted into a pandas data frame and visualized with the matplotlib package.

```
# extract the table containing daily infection data
peronospora_daily_table = list(filter(lambda table: table['Name'] == 'DailyOutput', response['Tables']))[0]

# tell pandas which are the fields containing column names and actual rows
peronospora_daily_df = pd.DataFrame(peronospora_daily_table['Rows'], columns = peronospora_daily_table['ColumnNames'])

# setting dates as index
peronospora_daily_df['Date'] = pd.to_datetime(peronospora_daily_df['Date'])
peronospora_daily_df = peronospora_daily_df.set_index('Date')

# avoid plotting zeroes event
peronospora_daily_df['PeronosporaPrimaria'].replace(0,np.NaN,inplace=True)
peronospora_daily_df
```

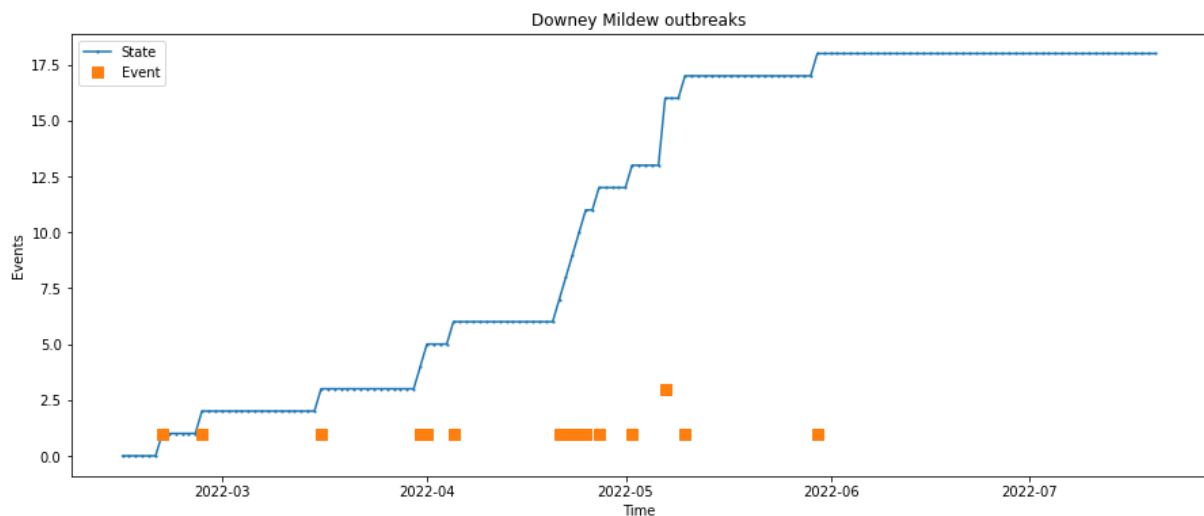
Date	PeronosporaPrimaria	PeronosporaPrimariaStato
2022-02-13 23:00:00+00:00	NaN	0
2022-02-14 23:00:00+00:00	NaN	0
2022-02-15 23:00:00+00:00	NaN	0
2022-02-16 23:00:00+00:00	NaN	0
2022-02-17 23:00:00+00:00	NaN	0
...	...	...
2022-07-15 23:00:00+00:00	NaN	18
2022-07-16 23:00:00+00:00	NaN	18
2022-07-17 23:00:00+00:00	NaN	18
2022-07-18 23:00:00+00:00	NaN	18
2022-07-19 23:00:00+00:00	NaN	18

157 rows × 2 columns

The NaN values in the table should be interpreted as no outbreak risk, while positive values indicate the number of events that may happen on that day, the higher, the more vigorous the infection could be. We can clearly see that by the

end of the considered period we would have 18 events, but to better understand how these events are distributed, we can plot them.

```
plt.figure(figsize=(15,6))
plt.title("Downey Mildew outbreaks")
plt.plot(peronospora_daily_df.index,peronospora_daily_df["PeronosporaPrimariaStato"],
marker='s', markersize=1, linestyle='-', label='State')
plt.plot(peronospora_daily_df.index,peronospora_daily_df["PeronosporaPrimaria"],
marker='s', markersize=8, linestyle='none', label='Event')
plt.ylabel('Events')
plt.xlabel('Time')
plt.legend()
plt.show()
```



## Powdery Mildew model

Powdery Mildew is another well-known one and we have a model also for that. The Web service is identical to the Downey Mildew one and we can simply call it attaching the very same request payload.

```
response_oidio = call_restful_api(
    'Oidio',
    data = model_data,
    method='POST')
```

The model responds with a table as well, containing disease outbreak events and cumulative events, just like the Downey Mildew model.

```
# extract the table containing daily infection data
oidio_daily_table = list(filter(lambda table: table['Name'] == 'DailyOutput', response_oidio['Tables']))[0]

# tell pandas which are the fields containing column names and actual rows
oidio_daily_df = pd.DataFrame(oidio_daily_table['Rows'], columns = oidio_daily_table['ColumnNames'])

# setting dates as index
oidio_daily_df['Date'] = pd.to_datetime(oidio_daily_df['Date'])
oidio_daily_df = oidio_daily_df.set_index('Date')

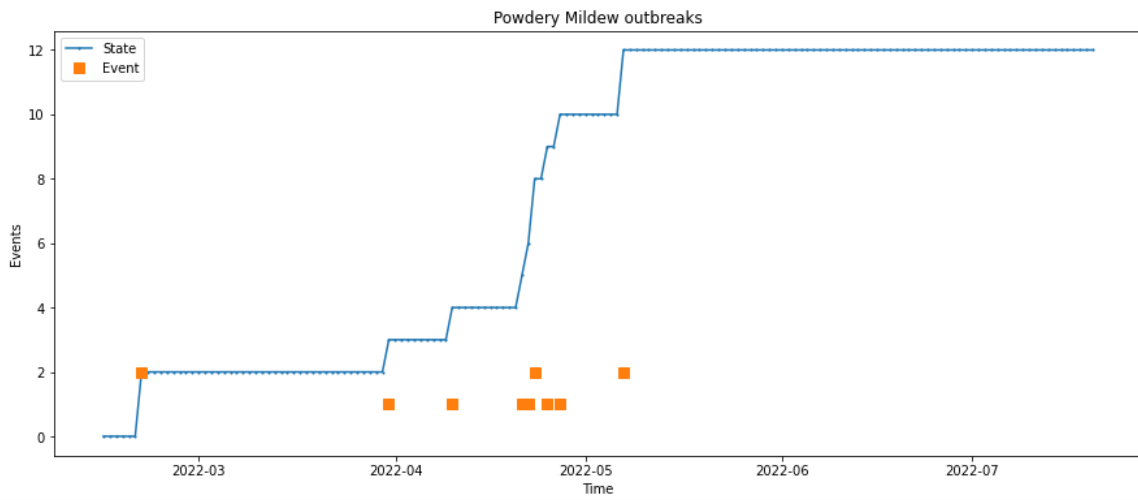
# avoid plotting zeroes event
oidio_daily_df['OidioPrimaria'].replace(0,np.NaN,inplace=True)
oidio_daily_df
```

Date	OidioPrimaria	OidioPrimariaStato
2022-02-13 23:00:00+00:00	NaN	0
2022-02-14 23:00:00+00:00	NaN	0
2022-02-15 23:00:00+00:00	NaN	0
2022-02-16 23:00:00+00:00	NaN	0
2022-02-17 23:00:00+00:00	NaN	0
...	...	...
2022-07-15 23:00:00+00:00	NaN	12
2022-07-16 23:00:00+00:00	NaN	12
2022-07-17 23:00:00+00:00	NaN	12
2022-07-18 23:00:00+00:00	NaN	12
2022-07-19 23:00:00+00:00	NaN	12

157 rows × 2 columns

We can plot of Powdery Mildew outbreaks.

```
plt.figure(figsize=(15,6))
plt.title("Powdery Mildew outbreaks")
plt.plot(oidio_daily_df.index,oidio_daily_df["OidioPrimariaStato"],
marker='s', markersize=1, linestyle='-', label='State')
plt.plot(oidio_daily_df.index,oidio_daily_df["OidioPrimaria"],
marker='s', markersize=8, linestyle='none', label='Event')
plt.ylabel('Events')
plt.xlabel('Time')
plt.legend()
plt.show()
```



## Weather Anomalies Checks and Corrections

In this example we show how to call the weather data quality check and correction API in a Python environment. The configuration (detectors and correctors) and the weather series to be corrected are already prepared in the folder "Files" with the filename "WeatherQuality\_configuration.json". This configuration will be used to prepare the Client with all the information to be passed to the service.

The following cell imports all the required dependencies.

```
# statistical processing components
import numpy as np
import pandas as pd

# presentation components
import matplotlib as mpl
import matplotlib.pyplot as plt
# this option allows plots to be embedded in the notebook file
%matplotlib inline

# http request related components
import requests
import json

# regular expressions
import re

#date and time
import datetime

#input and output Low level util
from io import StringIO

#OS utilities. We'll use it just to retrieve env variables.
import os
```

Now it is time to read from the environment variables the API key.

```
: # important variables
SERVICE_URL = 'https://api.progettoagridigit.it/weatherquality'
SERVICE_KEY = os.environ['AGRIDIGIT_API_KEY']
```

## Data Preparation

The Weather quality service was developed with the kernel BioMA, and it is part of a wide service ecosystem, hence it uses BioMA-like formats for data exchange.

### Weather data table

At its core, the system handles data in tables, and it expects each table to be described by four attributes:

1. `ColumnNames` which is a list containing columns names, it's the equivalent of csv file's header row.
2. `NotNullFields` which is the subset of the above list that identifies columns that have at least one non-null value.
3. `Rows` which contains the actual data expressed as a list of lists.
4. `Name` which contains the type of table, and it can be weather or locations, or any name that can be interpreted by the model sitting on the other side.

Each system table can be represented as a dictionary as follows:

```
{
  'ColumnNames': [list of weather variables labels],
  'NotNullFields' : [list of variables with at least an observation],
  'Rows': [
    [values for row 1],
    [values for row 2],
    ...
    [values for row n]
  ],
  'Name': 'table type'
}
```

These table objects must be wrapped in a Table attribute to make a valid data dictionary, so the result should look like this:

```
{
  'data':{
    'Tables':[
      {Data Table 1},
      {Data Table 2},
      ...
      {Data Table n}
    ]
  }
}
```

In addition, model data can include other information than data tables, like metadata or model parameters, in this case, it expects an additional configuration dictionary containing data quality check and gap filling rules. The full specification is a little verbose, but in a nutshell, it expects:

1. a type of attribute referencing the model object that will perform the data quality control.
2. a list of AnomaliesDetectors, i.e. objects that perform data quality control, with their respective parameters.
3. a list of AnomaliesCorrectors, i.e. objects that perform gap filling and other corrections, with their respective parameters.
4. the date/time combination on which the data series is supposed to begin.
5. the date/time combination on which the data series is supposed to end.
6. whether or not to perform a second data quality check after data correction.

A valid configuration object looks like this:

```
{
  'configuration': {
    '@type': 'BioMA.WeatherProviders.NetFramework.quality.correction.AnomaliesDetectAndFixComposer,
BioMA.WeatherProviders.Quality, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null',
    'AnomaliesDetectors':[...],
    'AnomaliesCorrectors' : [...],
    'StartDate': 'YYYY-MM-DDTHH:MM:SS',
    'EndDate': 'YYYY-MM-DDTHH:MM:SS',
    'ExecutePostQualityCheck': true|false
  }
}
```

Since it is a little verbose and building it can be quite difficult, we provide you with a ready made in the files folder. Being a dictionary, it is serialized as a JSON object, which can be easily parsed with the Python json module.

```
f = open(os.getcwd() + os.sep + 'Files' + os.sep + 'WeatherQuality_configuration.json')
input_data = json.load(f)
f.close()

weather_table = list(filter(lambda table: table['Name'] == 'weather', input_data['data']['Tables']))[0]
```

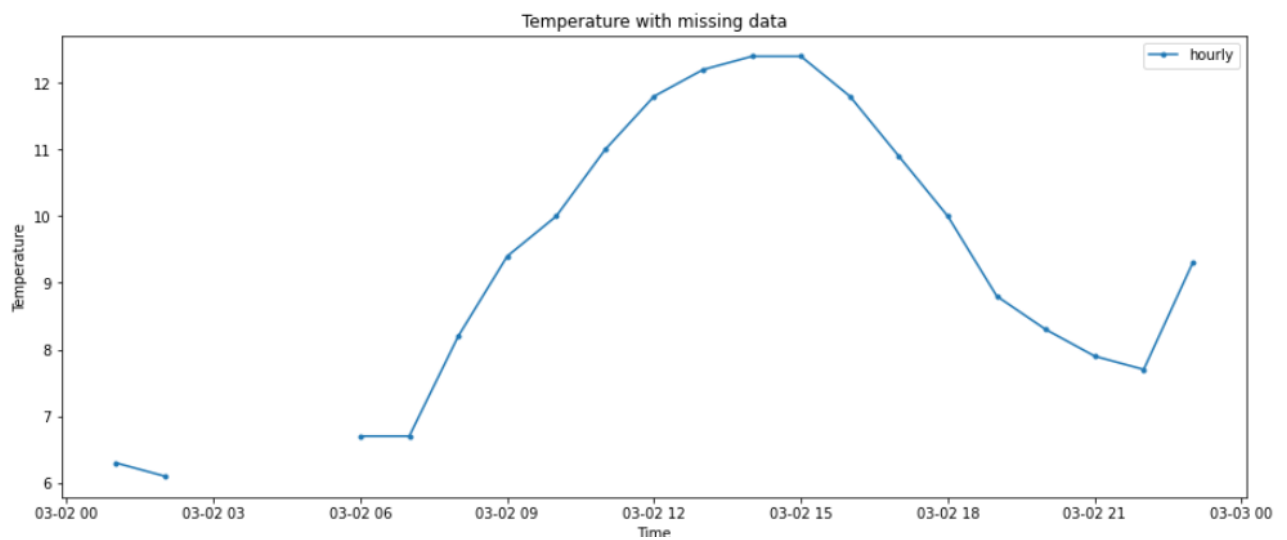
To better visualize the data in the sample file, we can convert the weather table into a pandas DataFrame. We can encapsulate this activity into a function to call it in multiple points of this notebook.

```
def weather_table_to_dataframe(weather_table):
    '''converts a BioMA data table into a pandas DataFrame object indexed by datetime'''
    # telling pandas which are the fields containing column names and actual rows
    weather_data_pd = pd.DataFrame(weather_table['Rows'], columns = weather_table['ColumnNames'])
    # parsing dates...
    weather_data_pd['Date'] = pd.to_datetime(weather_data_pd['Date'])
    # ... and setting them as rows index
    weather_data_pd = weather_data_pd.set_index('Date')
    # telling pandas the correct column type
    weather_data_pd['Average_temperature'] = weather_data_pd.Average_temperature.astype('float')
    return weather_data_pd
```

Now we can call it and plot the weather series with the Matplotlib Python package.

```
# call the function defined in the previous cell
weather_data_pd = weather_table_to_dataframe(weather_table)

plt.figure(figsize = (15, 6))
plt.title("Temperature with missing data")
plt.plot(weather_data_pd.index, weather_data_pd['Average_temperature'], marker='.', label = 'hourly')
plt.ylabel('Temperature')
plt.xlabel('Time')
plt.legend()
plt.show()
```



As you can see we have a time series with an evident gap, a problem we expect our data quality service to fix.

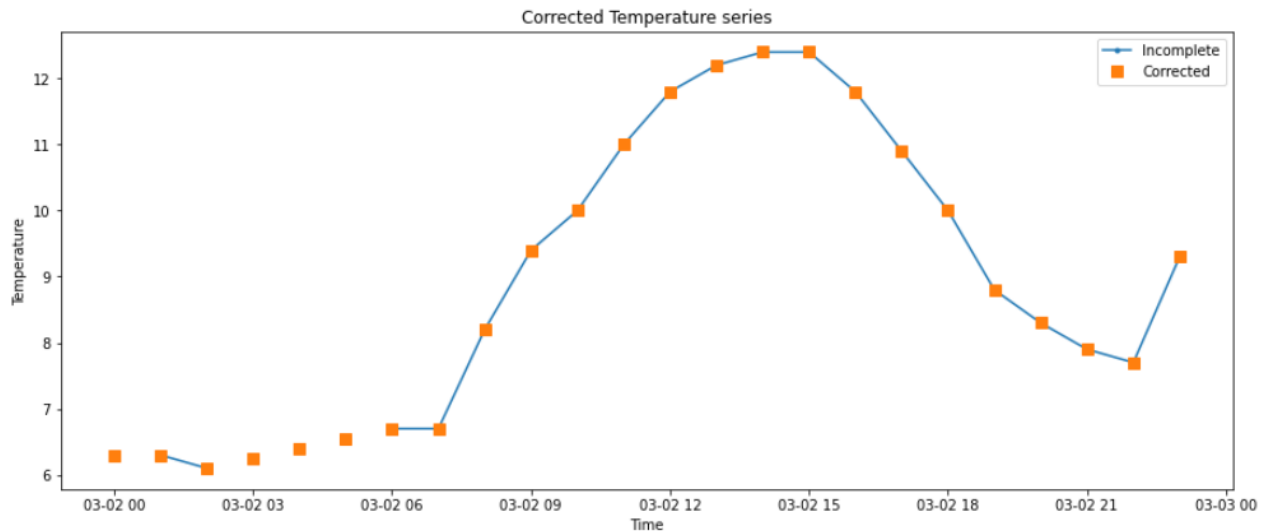
```
def call_restful_api(function, data={}, base_url = SERVICE_URL, key=SERVICE_KEY, method='POST'):
    # set request headers
    headers = {'Content-Type': 'application/json'}
    headers['Ocp-Apim-Subscription-Key'] = key
    # serialize request data into a json object
    test_data = json.dumps(data)
    response = None
    # invoke the Web Service
    if method == 'POST':
        response = requests.post(base_url + '/' + function, data=test_data, headers=headers)
    else:
        response=requests.get(base_url + '/' + function, data=test_data, headers=headers)
    if response.status_code == 200: # the 200 status means 'success'
        try:
            return response.json()
        except:
            try:
                return response.text
            except:
                print('Unable to decode server response')
    else:
        print('Request failed with code ' + str(response.status_code))
    return None
```

Now we call the service with the weather data and the configuration read from the local file, which will perform a linear interpolation of missing data points.

```
response = call_restful_api('Quality', data = input_data, method='POST')
```

A JSON file is returned that the requests module unpacks into a dictionary on its own, and it contains the corrected data in the Tables attribute. In the same way as for the input data, we can dump these data into a pandas data frame.

```
# pick only the weather table
corrected_weather_table = list(filter(lambda table: table['Name'] == 'weather', response['Tables']))[0]
# convert the BioMA table to a pandas dataframe
corrected_weather_data_df = weather_table_to_dataframe(corrected_weather_table)
# it's plottin' time
plt.figure(figsize=(15,6))
plt.title("Corrected Temperature series")
plt.plot(weather_data_pd.index,weather_data_pd['Average_temperature'],
marker='.', linestyle='-', label='Incomplete')
plt.plot(corrected_weather_data_df.index,corrected_weather_data_df["Average_temperature"],
marker='s', markersize=8, linestyle='none', label='Corrected')
plt.ylabel('Temperature')
plt.xlabel('Time')
plt.legend()
plt.show()
```



### A more advanced example

In the next example, we are correcting a Temperature series which has more missing data points, so that linear interpolation is no longer suitable. The flexibility of the weather data quality control service, however, has got us covered, as we can specify different policies in the configuration dictionary. In fact, we can declare a new configuration to include a fallback policy in case of large gaps in the data. Such a configuration is provided in the JSON configuration file, and its most notable feature are:

1. The Corrector `LinearInterpolationVariableHolesCorrection` is configured to trigger if no more than 4 hours worth of data are missing.
2. The `DerivedHourlyTemperatureCorrection` on the other hand has no such limit, so that corrector is applied and missing data points are filled by means of the `HATCampbell Clima` strategy.

Again, as the configuration file is rather verbose, we won't get into too much detail, but just read it from the file, along with some meaningful test data.

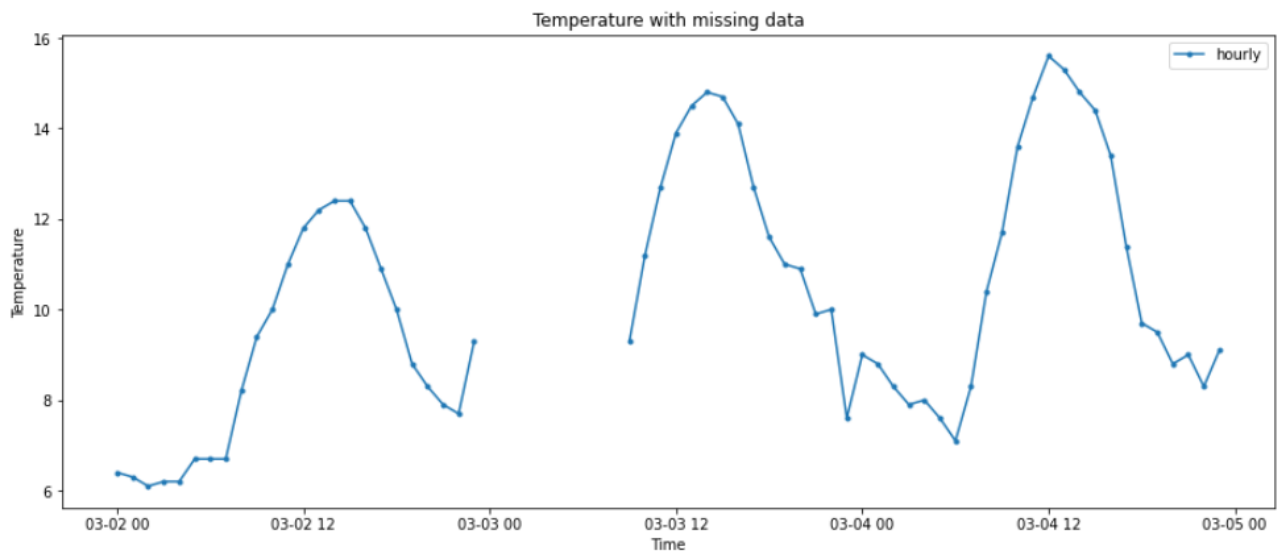
```
f = open(os.getcwd() + os.sep + 'Files' + os.sep + 'SeveralDaysWeatherQuality_configuration.json')
input_data_more_days = json.load(f)
f.close()
```

Let's visualize the sample data.

```
weather_table_more_days = list(filter(lambda table: table['Name'] == 'weather', input_data_more_days['data']['Tables']))[0]

weather_data_more_days_pd = weather_table_to_dataframe(weather_table_more_days)

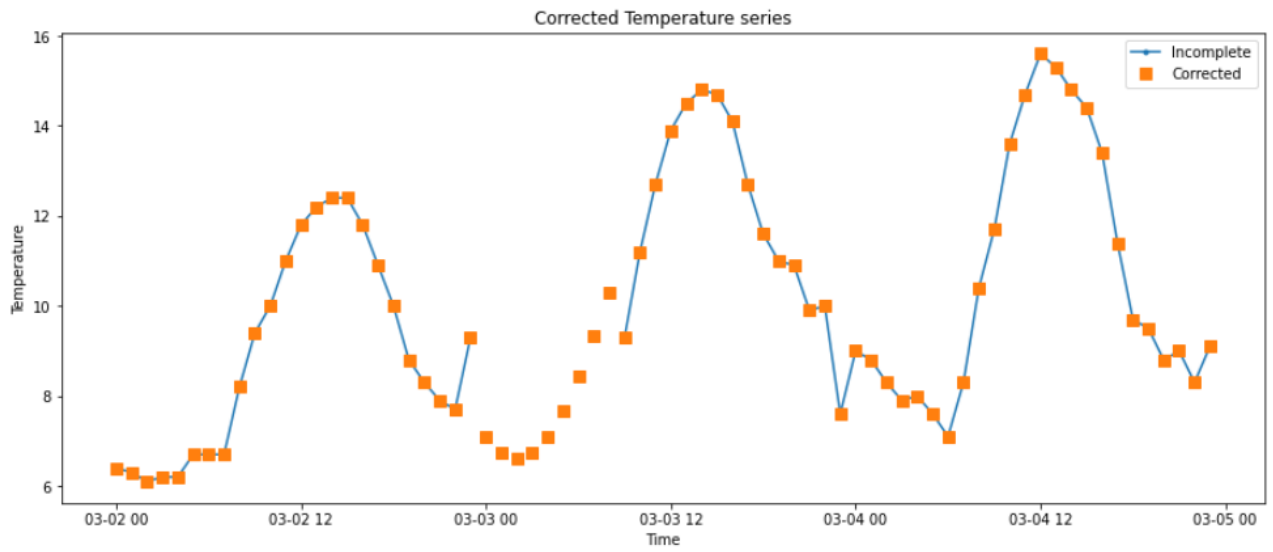
plt.figure(figsize = (15, 6))
plt.title("Temperature with missing data")
plt.plot(weather_data_more_days_pd.index, weather_data_more_days_pd['Average_temperature'], marker='.', label = 'hourly')
plt.ylabel('Temperature')
plt.xlabel('Time')
plt.legend()
plt.show()
```



As you can easily notice, a large chunk of data is missing, and we expect such gap to be filled in a nonlinear way, as the surrounding series clearly exhibits a periodic trajectory. Let's invoke the Web service and see what happens.

```
response_more_data = call_restful_api('Quality', data = input_data_more_days, method='POST')
```

```
# pick only weather table
corrected_weather_table_more_days = list(filter(lambda table: table['Name'] == 'weather', response_more_data['Tables']))[0]
# BioMA to dataframe conversion
corrected_weather_data_table_more_days_df = weather_table_to_dataframe(corrected_weather_table_more_days)
plt.figure(figsize=(15,6))
# plotting logic
plt.title("Corrected Temperature series")
plt.plot(weather_data_more_days_pd.index,weather_data_more_days_pd['Average_temperature'],
marker='.', linestyle='-', label='Incomplete')
plt.plot(corrected_weather_data_table_more_days_df.index,corrected_weather_data_table_more_days_df["Average_temperature"],
marker='s', markersize=8, linestyle='none', label='Corrected')
plt.ylabel('Temperature')
plt.xlabel('Time')
plt.legend()
plt.show()
```



## Image classification services

This example shows how to interact with computer vision services. We assume that you have already set up properly your environment, including both dependencies and global variables, if you have not done it yet, check out the first example in this series and follow the procedure therein described.

The following cell imports all the required dependencies.

```
# Image manipulation related components
from PIL import Image
import numpy as np

# http request related components
import requests
import json

# I/O components
import os
from io import BytesIO
import base64

# presentation components
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Now it is time to read from the environment variables the API key. If you ran the previous example, it should be already set up properly. We also declare a service access function, encapsulating all the http request and exception handling.

```
# important variables
SERVICE_URL = 'https://api.progettoagridigit.it/bee-wings-classifier'
SERVICE_KEY = os.environ['AGRIDIGIT_API_KEY']

def call_restful_api(function, data={}, base_url = SERVICE_URL, key=SERVICE_KEY, method='POST'):
    # set request headers
    headers = {'Content-Type': 'application/json'}
    headers['Ocp-Apim-Subscription-Key'] = key
    # serialize request data into a json object
    test_data = json.dumps(data)
    response = None
    # invoke the Web Service
    if method == 'POST':
        response = requests.post(base_url + '/' + function, data=test_data, headers=headers)
    else:
        response=requests.get(base_url + '/' + function, data=test_data, headers=headers)
    if response.status_code == 200: # the 200 status means 'success'
        try:
            return response.json()
        except:
            try:
                return response.text
            except:
                print('Unable to decode server response')
    else:
        print('Request failed with code ' + str(response.status_code))
    return None
```

## Classifying grapevine leaf images

This API can provide an example in the form of the Grapevine Leaf classifier, which is able to identify the variety of a grapevine plant given a full-frontal leaf image.

```
LEAVES_SERVICE_URL = 'https://api.progettoagridigit.it/vine-varieties-classifier'
```

To demonstrate the system, we'll consider four images taken from the internet, and send them to the Web service.

```
image_names = ['cabernet.jpg', 'merlot.jpg', 'vermentino.jpg', 'pinot.jpg']
images = []
for n in image_names:
    images.append(Image.open('Files' + os.sep + n))

response = get_classification(images, image_names, function = 'score', base_url =LEAVES_SERVICE_URL, key= SERVICE_KEY)
response
```

As this model has substantially more classes, reading the output dictionary is extremely hard, hence it is important to visualize the classification results.