Project Report

DS50 – P24

Yann Derré | Pierre-Olivier Cayetanot | AUGUSTE Nicolas | HAO Zhihan | HU Zhuxin | KE Wentao

utbm
université de technologie Belfort-Montbéliard

INTRODUCTION

Satellite images are a key data source to understand events occurring on earth. Combining it with external sources like weather and atmospheric conditions for a given time allows data scientists to infer life-saving predictions about natural catastrophes.
For example, we could predict the spread of a wildfire or flood to evacuate an occupied area

It can, however, be difficult to work with, as they don't utilize familiar or modern data formats and have specific attributes when working with Earth's coordinates (coordinate systems, projection types...).

Obtaining the data in the first place can also be a difficult task, as those systems often require specific configurations to obtain the desired product.

The aim of this project is to streamline the process of acquiring and processing data from multiple sources and standardize it into an easy-to-use format for further data science applications.

The source code of the project is available on GitHub:
https://github.com/derreyann/sentinel-utbm

I – Project overview

In this first section, we will go over each component of the project, from the general project structure to the individual modules created to handle data sourcing and processing as well as visualization and exporting.

Project Structure

In this section, we will go over how the project is structured and get a high-level view of the different data flows and processes needed to achieve our goals.

a) Repository structure

The notebooks illustrate the usage of the different scripts holding our data fetching and processing functions.
The data folder allows you to cache results to your system. This folder path is customizable.

The project is structured with the following folders:

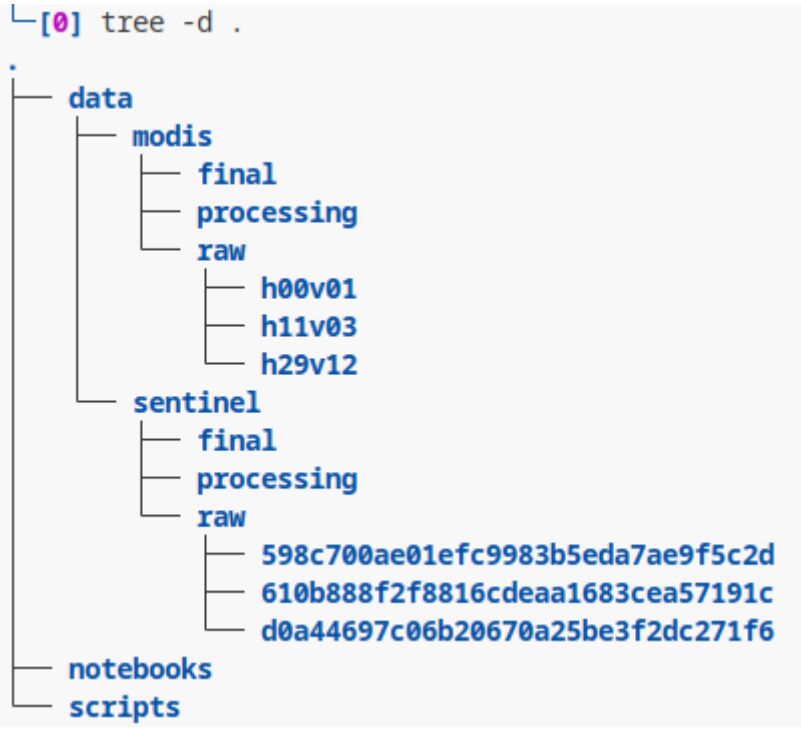
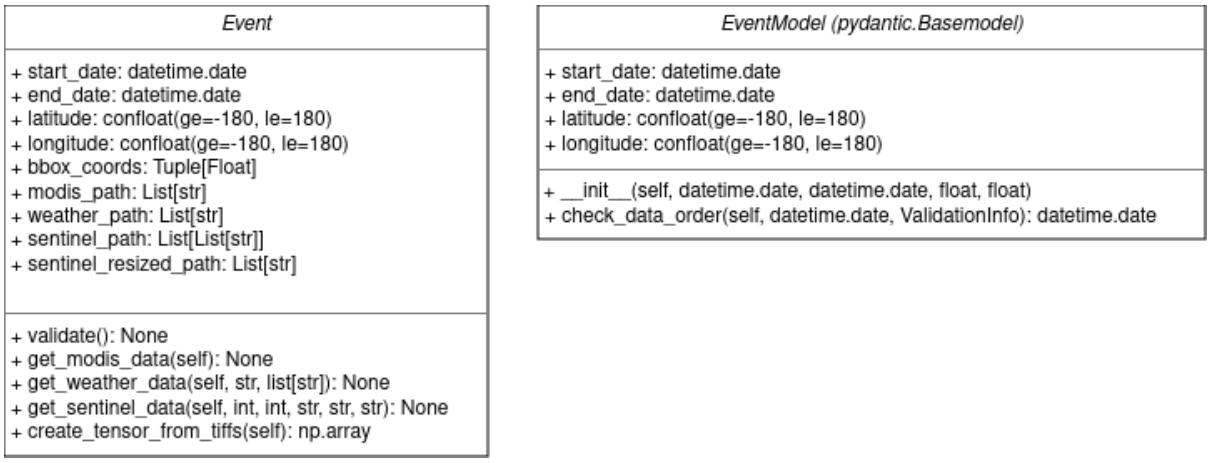


Fig. 1 Project folder structure

b) Event class

The event class easily gathers data from our sources, automatically applies basic preprocessing steps and enables users to retrieve tensors representing our event.
The notebook struct_usage.ipynb details the usage of the Event class.

ULM diagram



The *Event* model class uses *Pydantic* to validate user input using the *EventModel class*. It inherits it from the *Pydantic.BaseModel* class to parse and validate input.

The Event class retrieves and preprocess data from our three data sources and sets the path to those resources in the class. Finally, you can create a tensor from the available data that was previously set using the other functions.

Usage

Our case study will further explain how the different functions are used, visualization and how go further.

The class allows users to retrieve the paths to the preprocessed data.

The data is split by 8 days interval, and each element of the path array represents one interval.

The weather data represents multiple features at a time: *wind speed, rain…* Each feature is saved to a different file, thus the usage of List[List[str]].
Those files are opened using the *rasterio library*, allowing us to interact with GDAL for opening and rasterizing file.

```
# Open the first available file
modis = rasterio.open(event.modis_path[0])
ep.plot_bands(modis.read(1), title="Fire", cmap="YlOrRd", figsize=(10, 10))
```
✓ 0.1s

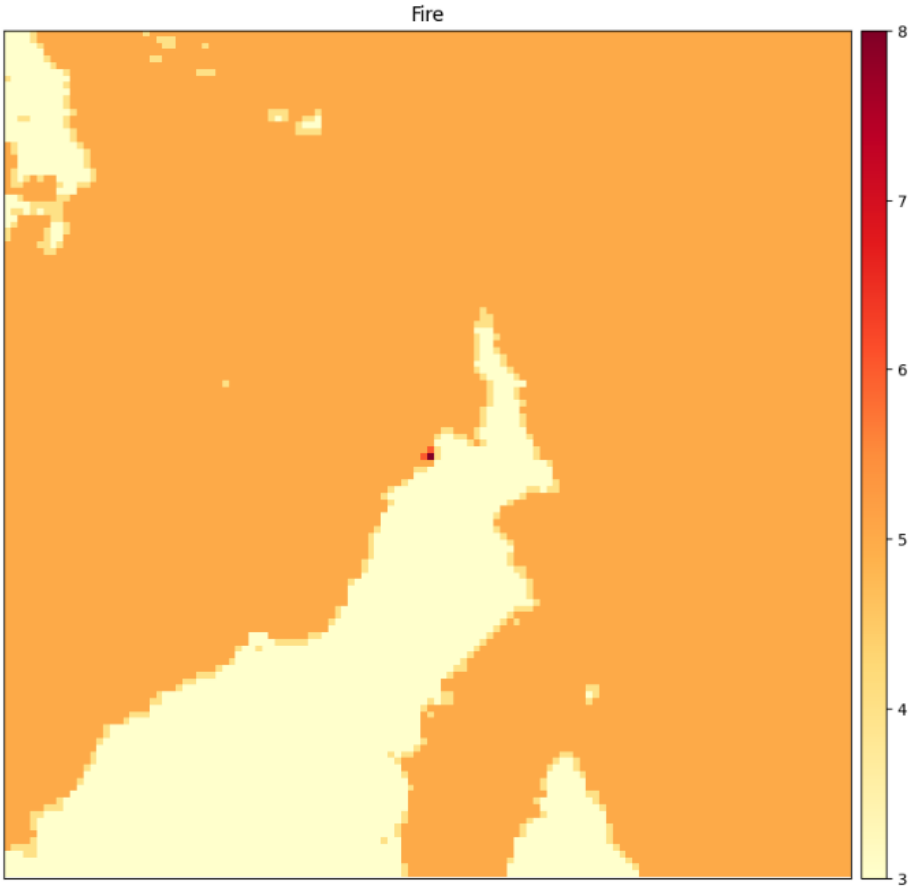


Fig. 2 – Retrieving a fire mask

The tensor output from the class depends on the retrieved data:

```
tensor = event.create_tensor_from_tiffs()
tensor.shape
```
✓ 0.0s

```
(1, 10, 128, 128)
```

The shape of the tensor will depend on the number of weather features taken, the number of 8-day periods and the number of bands used by the evalscript for sentinel data.
In this example, we have 4 bands, 5 weather features and modis, giving us a final tensor of shape (1, 10, 128, 128): 1 8-day interval, 10 features for a 128 by 128 resolution.

c) Data flows

General data flow

The project uses three data sources: SentinelHub, Modis and MeteoStat.

Those data sources and their manipulation are described thoroughly in their dedicated sections.
This section shows the larger ETL steps and logic used to retrieve, process and save data to enable users to easily create tensors ready to be fed into models or further analyzed.

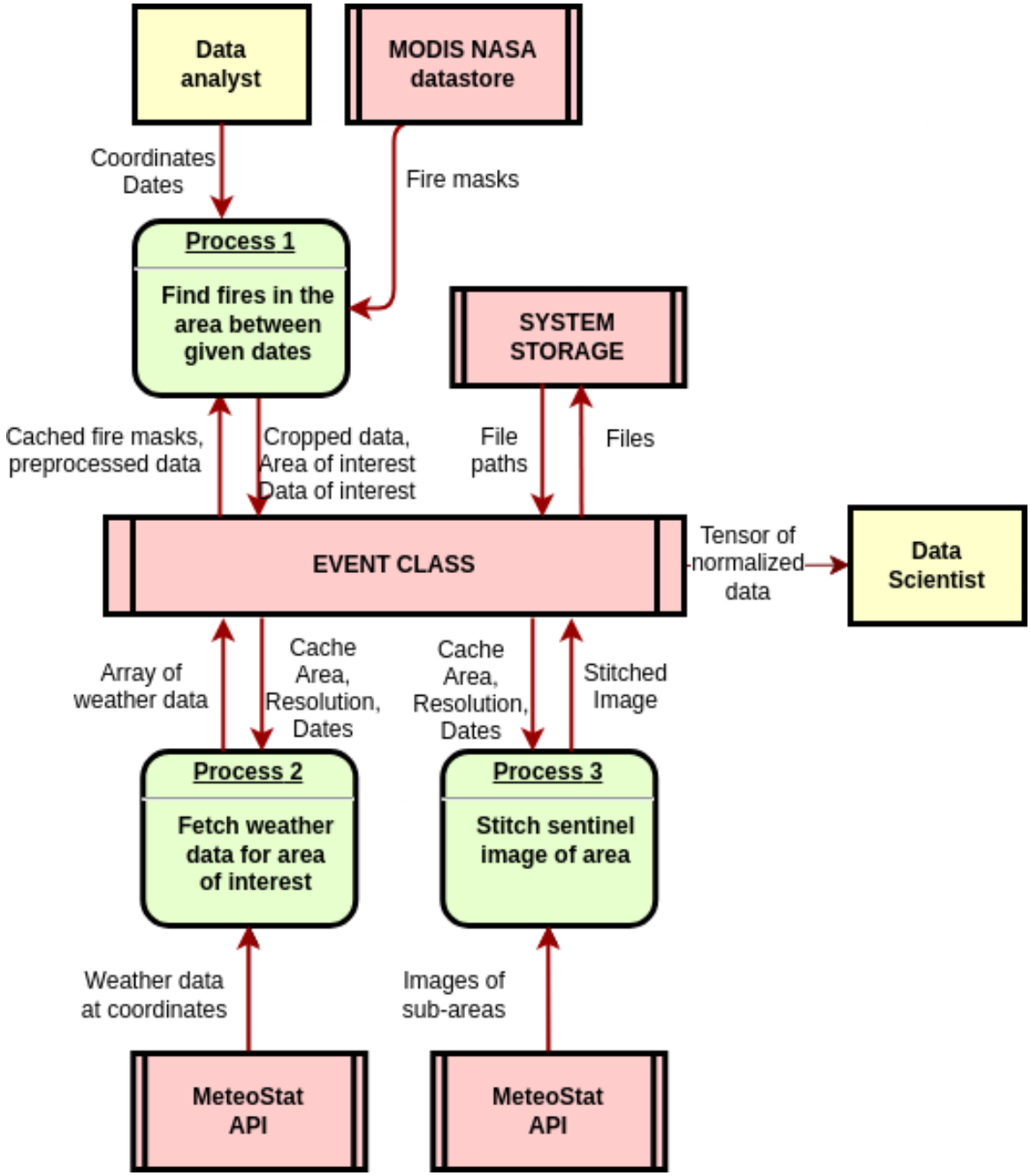


Fig. 2 General dataflow

Modis Data Processing

The modis data is used to set parameters used by the other data fetchers. It answers the following questions:

- ***Time**: When is the first fire occuring at that location during that time span?*
- ***Area**: Where is the fire occurring?*
- ***Mask**: What is the state of that area?*

When two, time disjointed fires occur, we only retrieve and preprocess the first one: we stop the preprocessing if no fire is detected in the next file (8-day period).

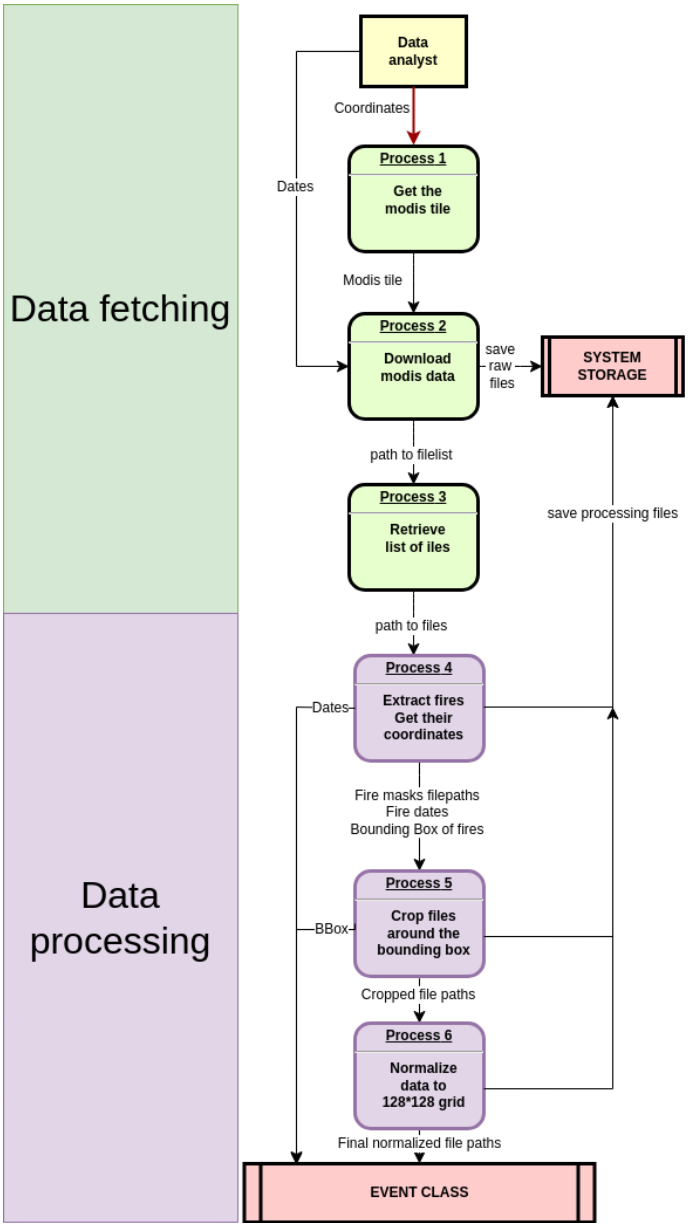


Fig.3 Modis data flow

Weather data

After computing our area of interest using MODIS data, we can easily fetch weather data using the final normalized file: it contains a 128 by 128 grid of localizations.

For each of these coordinates, we fetch a set of weather parameters. We have a daily time resolution, and the data is fetched by batches of 5 days, to match the Sentinel periodicity.

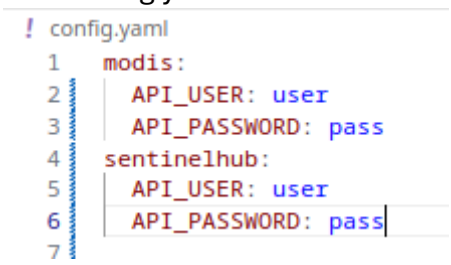
Sentinel data

For each 8-day Modis period, we stitch together an image of the area using the SentinelHub API.

The Sentinel satellites we are using have a 5-day periodicity. As such, we either only use the available images, or stitch together the best of the two batches available over the 8-day period.

Credentials

The credentials are stored in the config.yaml file.



MODIS Data

To accurately obtain labels for our fire data, we utilize data from a Modis Satellite Product, MOD14A1 – “Thermal Anomalies/Fire Daily”. The data derived contains information collected daily by the MODIS Terra instrument and published in archives of 8 days.

It consists of 9 different masks, 3 of them dedicated to Fire Predictions with the layers 7 to 9 consisting of low-medium-high predictions respectively.

The archives are stored in the HDF4 format, a legacy file format chosen by NASA back in 1994. One of the challenges was setting-up support for this format, as most libraries and dependencies are not supporting it out-of-the-box anymore.

In this section, we’ll go into details concerning the system setup-used in our project.

d) Dependencies

PyMODIS is the Python library we chose for our project to directly fetch the Terra satellite data from the NASA Earthdata's servers. It provides an easy way for us to :

- Specify a date-range
- Specify a product type gere *(here MOD14A1, but other products could be utilized for water detection for example)*
- Geographical tile

NASA Account Creation + pymodis

Although the files on the server we're downloading from (https://e4ftl01.cr.usgs.gov/) are downloadable without login, the PyMODIS library requires us to create an Earthdata account. There are virtually no limitations with this account as the data is public.

- Create an account with is URL : https://urs.earthdata.nasa.gov/

- Inside the downModis() function, fill in your user and password variables with your credentials.

You're all done!

GDAL Setup

In our project *GDAL (Geospatial Data Abstraction Library)* plays a critical role in processing and managing MODIS HDF4 files. It actively acts as the backend for most of the Python libraries we're using to read, and transform our data files.

These libraries create bindings to GDAL during install, so make sure to install GDAL first, then the python requirements. You might need to reinstall your python requirements if HDF4 support/GDAL is not detected.

Let's install GDAL:
On macOS using MacPorts:
sudo port install gdal +hdf4

On Ubuntu/Debian using apt:
sudo apt install libhdf4-0-alt libhdf4-dev

On Windows, *a binary can directly be found on the GDAL website:*
https://gdal.org/

To verify if HDF4 support has been successfully enabled in GDAL, you can run this command:
gdalinfo --formats | grep HDF4

If everything's in order, you are now ready to install the Python environment.

Python environment Setup

We used a conda environment for our project containing all of our libraries. You should find a file called environment.yml file.

You can import our environment using this command
conda env create -f environment.yml

A poetry environment is also available using:
poetry install

You will need to install rasterio from source manually for HDF4 support:
pip install rasterio==1.3.10 --no-binary :all:

You're ready to launch our Python scripts and notebooks!

Downloading MODIS Data

The MODIS products utilize a Sinusoidal Grid System. This grid divides the Earth into tiles, each measuring approximately 10 degrees by 10 degrees at the equator. The grid is defined by horizontal (h) and vertical (v) tile numbers, starting from (0,0) at the upper left corner and increasing to the right and downward.

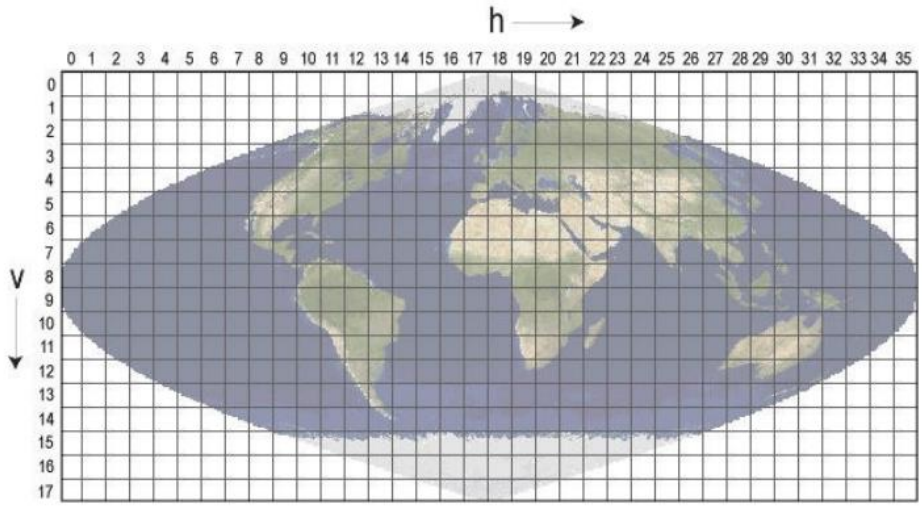


Figure 2. MODIS Sinusoidal Tiling System

From https://modis-land.gsfc.nasa.gov/pdf/MOD15_C6_UserGuide_June2020.pdf

We created a *get_tile()* method inside the *modis.py* file to provide an easy way to find which tile should be downloaded for a given geographical coordinate.

By default, our script saves that data inside the "data/modis" directory.

e) Data Analysis

To read the HDF file, we use a combination of two libraries, rasterio and rioxarray. To open the file, use the method open_rasterio(file), inside rioxarray.

Data Format

The data is stored inside an xarray.Dataset type, a structured file containing different masks as well as metadata about the granule.

```
<xarray.Dataset> Size: 92MB
Dimensions:      (band: 8, x: 1200, y: 1200)
Coordinates:
  * band         (band) int64 64B 1 2 3 4 5 6 7 8
  * x            (x) float64 10kB -8.895e+06 -8.894e+06 ... -7.784e+06
  * y            (y) float64 10kB 6.671e+06 6.67e+06 ... 5.561e+06 5.56e+06
    spatial_ref  int64 8B 0
Data variables:
    FireMask     (band, y, x) uint8 12MB ...
    QA           (band, y, x) uint8 12MB ...
    MaxFRP       (band, y, x) int32 46MB ...
    sample       (band, y, x) uint16 23MB ...
Attributes: (12/84)
    ALGORITHMPACKAGEACCEPTANCEDATE:     04-2006
    ALGORITHMPACKAGEMATURITYCODE:       Normal
    ALGORITHMPACKAGENAME:               MOD_PR14A
    ALGORITHMPACKAGEVERSION:            5
    ASSOCIATEDINSTRUMENTSHORTNAME.1:    MODIS
    ASSOCIATEDPLATFORMSHORTNAME.1:      Terra
    ...                                 ...
    TileID:                             51010003
    UnknownPix:                         0, 0, 0, 0, 1, 14, 2, 5
    VERSIONID:                          61
    VERTICALTILENUMBER:                 3
    VerticalTileNumber:                 3
    WESTBOUNDINGCOORDINATE:             -159.99999995957
```

An example of an HDF file read by rioxarray.

Amongst the interesting metadata, we find a data array containing the 8 different days inside the bands. Each band corresponds to one day of capture.

To properly plot our data, we need to reproject the data from the MODIS Sinusoidal projection to a standard Mercator projection **WGS 84**, the *World Geodetic System* also called ***EPSG:4326*.**

Fortunately, *rasterio* contains a method for this:

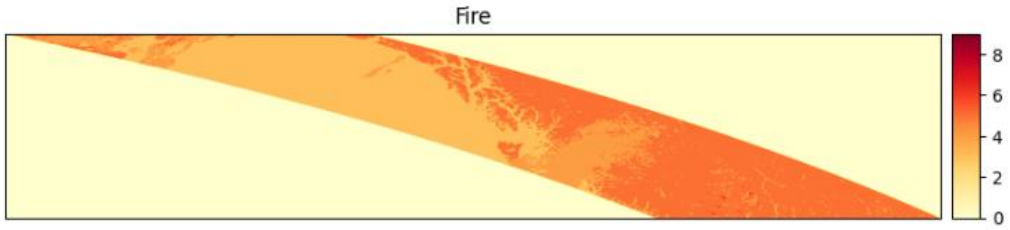
```
tiff_file = tiff_file.rio.reproject("EPSG:4326")
```

We chose to save the data as a .tif file for further use to standardize our data types across the project, using the to_raster() method. We are now ready to plot the data.

Plotting the data

The mask containing the fire prediction data is contained in the *FireMask layer*. The *earthplot* library is used to generate representations of the special data

```
In [89]: ep.plot_bands(fire.read(1), title="Fire", cmap="YlOrRd", figsize=(10, 10))
```



Here's an example of a granule (h10v03) reprojected. The read(1) reads the first band of the dataset, here the first out of the 8 days.

Extracting the fire data

Once the data is reprojected, we can have an equivalent pixel to geographical coordinates, meaning that we can extract the location of the Mask pixel between 7-9 corresponding to the predicted fires.

Using the .xy(X, Y) method, we can give a X and Y pixel coordinate from the image and get its (latitude, longitude) tuple as an output.

The *get_coords_and_pixels(dataset)* method was built to easily extract this from a given Mask.

```
In [92]: coords, pixels = get_coords_and_pixels(fire)
```

```
Fire Pixel at:  (-120.95749637234849, 51.91536369187604)
Fire Pixel at:  (-119.730227104381, 51.70059156998173)
Fire Pixel at:  (-119.69954537268181, 51.70059156998173)
Fire Pixel at:  (-122.85976373769807, 51.57786464318498)
Fire Pixel at:  (-117.52114242203953, 51.57786464318498)
Fire Pixel at:  (-121.54044927463303, 51.5471829114858)
Fire Pixel at:  (-121.69385793312897, 51.42445598468905)
Fire Pixel at:  (-121.60181273803141, 51.39377425298986)
Fire Pixel at:  (-121.57113100633222, 51.363092521290675)
```

Cropping the study zone

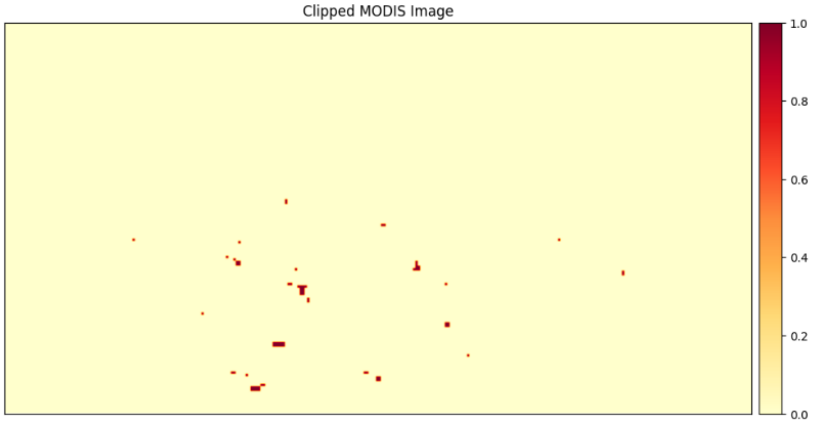
To crop the data around the study zone, we provide a script to generate a bounding box around the fire pixel, with added padding.

We must modify the .tif profile to crop the visualization window.

```
# Get the profile of the cropped data
profile = src.profile.copy()
profile.update(
    {
        "height": window.height,
        "width": window.width,
        "transform": src.window_transform(window),
    }
)
```

We then pass this profile as an argument to *Rasterio:*

```python
# Write the cropped MODIS data to a new GeoTIFF file
with rasterio.open(modis_clipped_path, "w", **profile) as dst:
    dst.write(modis_data)
```



The image is now cropped.

Weather Data

We used the *meteostat* library to fetch weather data across the studied zone. For each coordinate extracted from the pixels, we fetch an interpolated weather data. Amongst the available data points, we selected 4:

- *Average temperature*
- *Average precipitation*
- *Wind speed*
- *Wind direction (degrees)*

f) Data Sources

Meteostat uses weather data provided by the following organizations:

- Deutscher Wetterdienst
- NOAA - National Weather Service
- NOAA - Global Historical Climatology Network
- NOAA - Integrated Surface Database
- Government of Canada - Open Data
- MET Norway
- European Data Portal

g) Data Processing

Outside of the interpolation done server-side, we convert the wind direction from degrees to cos/sin to properly represent the circular nature of the direction e.g. 359 degrees has to be a similar value to 2 degrees.

We gather the data for the 8 bands contained in the dataset, corresponding to the 8 different days.

That data is stored in an empty tensor with the same format and profile as the MODIS cropped TIF file. We can thus use that weather data as a Mask, with a corresponding pixel-coordinate equivalent.

We choose to save the weather data as TIF files for further use.

Sentinel API

The Sentinel project is a European Union's Copernicus program, which is dedicated to Earth observation and monitoring.

It's comprised of a series of satellites, each equipped with specialized instruments to capture various imagery and data across various spectral bands. The data produced by those instruments is catalogued and is freely made available online.

In the section below, we'll explain which data we opted to use and why, as well as the steps necessary to gather and process that data.

h) Fetching satellite data

The goal of the project was to conceive an easy way to interact with the Sentinel satellite products for further use (fire, flood prediction, land classification…)

Although the gathered data could be used for many types of disaster, our focus was mainly on forest fires for the following reasons:

Sentinel-2 produces high-quality color satellite images, but with a temporal resolution of 5-days, meaning the instrument passes the same site 5 days later.

Forest fires are unfortunately lasting events compared to an industrial disaster, as well as having the potential of being spread across a much wider area.

Our case study spans for example across many months on the southern part of the eastern Australian coast. This made analysis and time resolution standardization much easier.

Types of satellite products

Two options were at our disposal:

- **Sentinel-5, which provides** products focused on atmospheric composition monitoring like carbon monoxide, useful for detecting emissions caused by combustion during fires. Those products have a 24-hour time resolution.

- **Sentinel-2, which provides** high-resolution satellite products, available in multiple wavelengths bands. One processing filter, “NDVI”, allows us to trace vegetation health, which is really useful for forest fire analysis. This instrument only produces one product every 5 days, which was one of the main constraints of the project.

The NDVI index provides useful information about vegetation health

We opted for the second option, as using high quality color images was more practical for future analysis, compared to the lower resolution Sentinel-5 images. This however meant that we would need to mitigate the lower temporal resolution of Sentinel-2

Connecting to the Sentinel API

An official API provides data directly from the Copernicus Servers: *https://dataspace.copernicus.eu/analyse/apis/sentinel-hub*

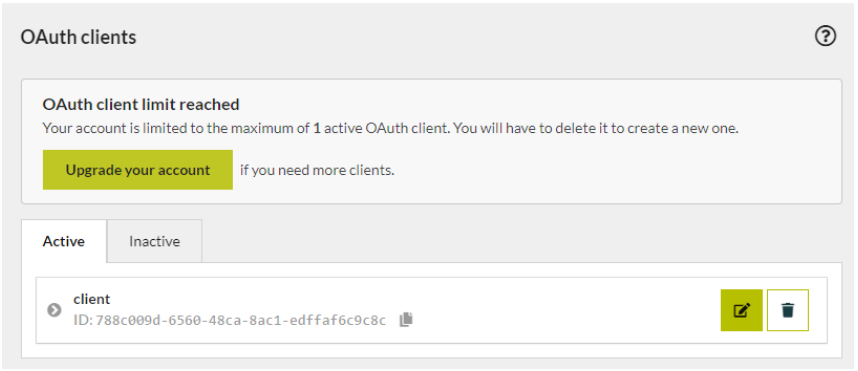
Unfortunately, we were not able to use the first-party API, as recent changes in the system broke the libraries currently available for Python.

Awaiting community fixes, we decided to opt for a third-party API, Sentinel Hub, operated by a private company Planet Labs: https://www.sentinel-hub.com/develop/api/

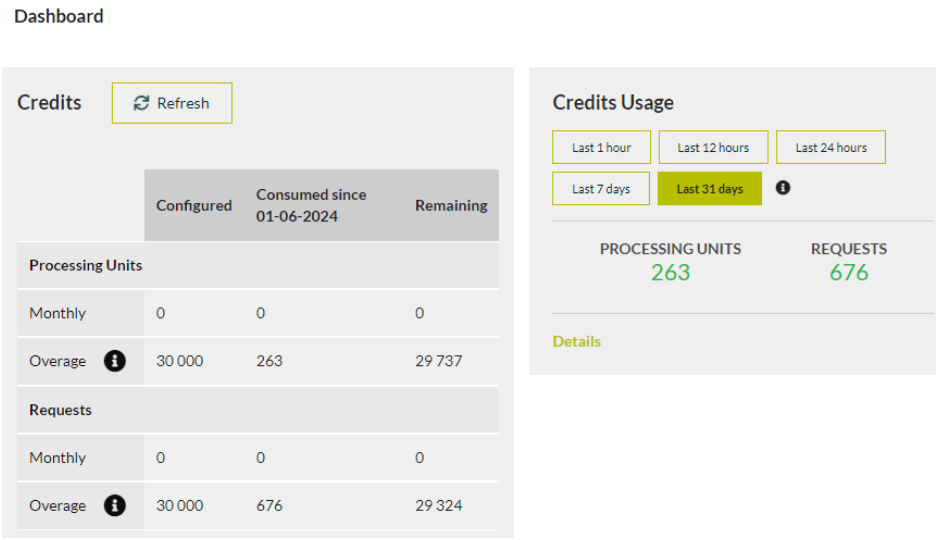
Fortunately, those two APIS have similar limitations and dashboards, meaning future transition to an official API could be easily done.

Here's a quick step-by-step usage of the API dashboard and integration into our project

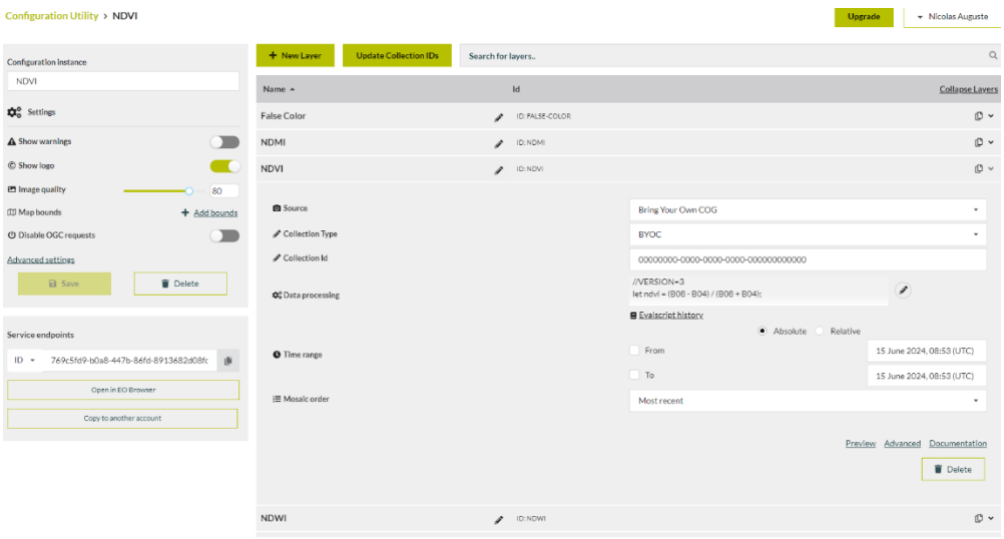
On the Dashboard, we had to create OAuth client credentials to connect our Python project to the API:



On the **Dashboard** page we can also see our use credit usage:



Finally on **Configuration Utility** page we can see custom creation we can use in our code or in other tools like QGIS:



Inside the project, we insert our credentials inside the config structure, allowing us to pass it as argument in the library functions.

Using sentinel API

After connecting our project to the API we can finally fetch the images produced from the satellite.

Let's take a look at the structure of a request to understand how sentinel API work:

```
request_ndvi_img = SentinelHubRequest(
    data_folder=data_folder,
    evalscript=evalscript_ndvi,
    input_data=[
        SentinelHubRequest.input_data(
            data_collection=DataCollection.SENTINEL2_L2A,
            time_interval=(start_date, end_date),
            other_args={"dataFilter": {"mosaickingOrder": "leastCC"}},
        )
    ],
    responses=[SentinelHubRequest.output_response("default", MimeType.TIFF)],
    bbox=aoi_bbox,
    size=aoi_size,
    config=config,
```

- The **bbox** parameter is the area of interest we defined.
- The **size parameter** is the resolution of the image gathered.
- We give an **output directory** to gather our generated images.
- We add an **EvalScript** (or "custom script"), a piece of JavaScript code which defines how the satellite data is processed server-side by Sentinel Hub and what

values the service should return. This is how a filter such as NDVI is generated. This however counts towards our *Processing* credits.

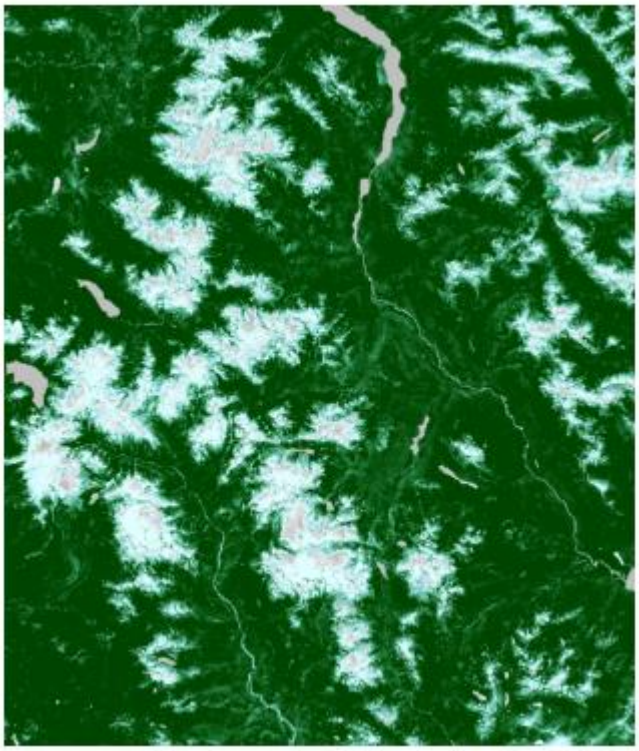
- We can also choose which **data collection** or satellite we are going to use for this request.

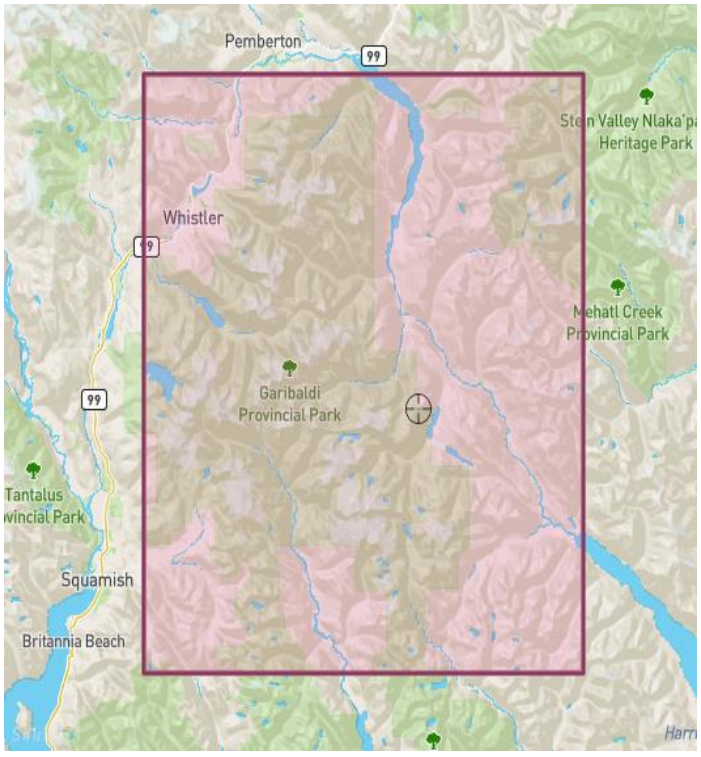
- A time interval is also selected but can be slightly shifted in case of low-quality data present in the timeframe. This can be disabled.

- We select extra arguments for later processing **least cloud coverage** and **mosaickingOrder** (Sets the order of overlapping tiles from which the output result is mosaicked).
- In the last part of the request, we use **MimeType.tiff** to download data as .tiff files.

i) Stitching satellite images together - API Resolution limitation

We're now able to request data from a given bounding box (or bbox). We can use websites like http://bboxfinder.com/ to easily represent them:





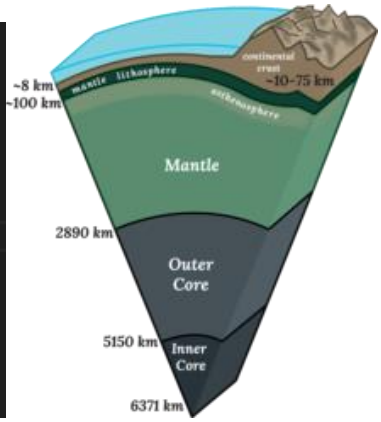
One of the main roadblocks was the total resolution of the studied areas, which required gathering very **large images**. The API however has a maximum 2500 x 2500 pixel resolution limit for each image.

To solve this, we developed a patching function which takes a large bbox and divides it in a list of point separated by the same distance to create smaller bboxes sent respectively to the API.

For this to work it was important to take in consideration the Earth's radius at various latitudes to connect perfectly every image. We used the standard **WGS84** coordinate system to facilitate this.

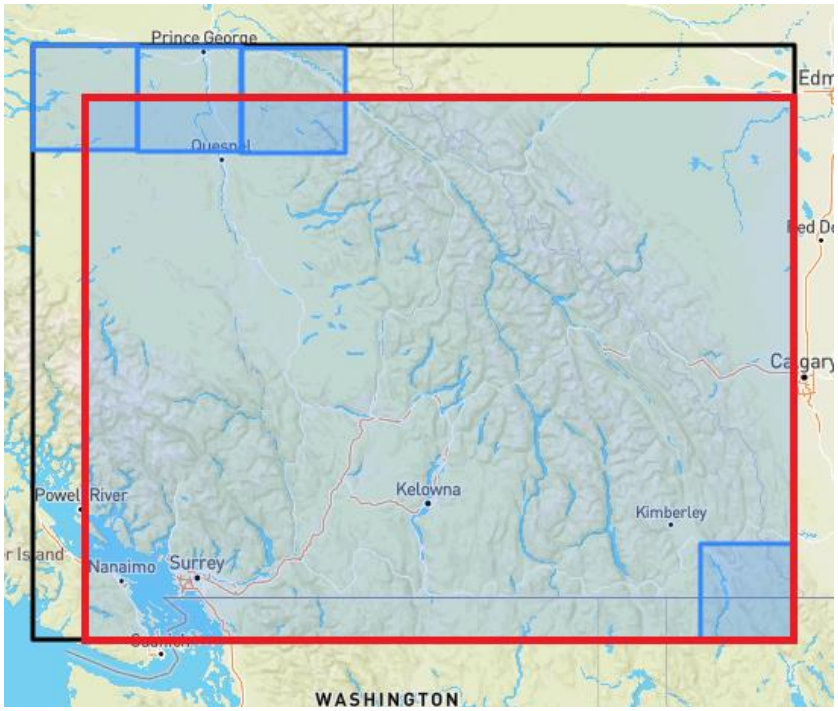
```python
# Semi-axes of WGS-84 geoidal reference
WGS84_a = 6378137.0  # Major semiaxis [m]
WGS84_b = 6356752.3  # Minor semiaxis [m]

# Earth radius at a given latitude, according to the WGS-84 ellipsoid [m]
def WGS84EarthRadius(lat):
    # http://en.wikipedia.org/wiki/Earth_radius
    An = WGS84_a*WGS84_a * math.cos(lat)
    Bn = WGS84_b*WGS84_b * math.sin(lat)
    Ad = WGS84_a * math.cos(lat)
    Bd = WGS84_b * math.sin(lat)
    return math.sqrt( (An*An + Bn*Bn)/(Ad*Ad + Bd*Bd) )
```



The WGS84 coordinate system implemented in code.

This is a visualisation of how the division of the large bbox works:



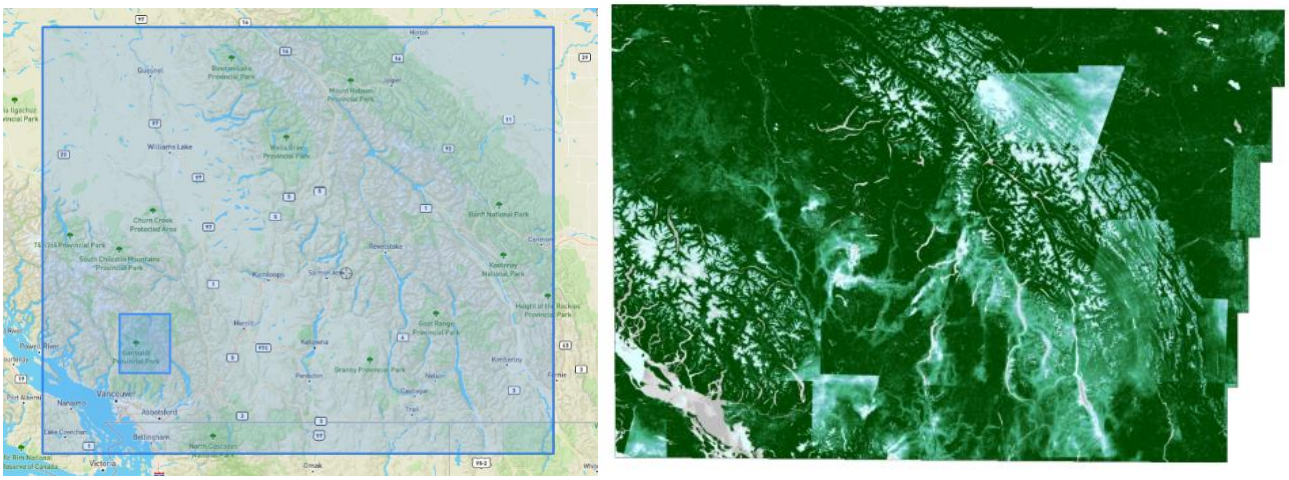
The orginal bbox (in red) and the generated smaller bboxes (in blue)

The subdivision of the bounding boxes is dictated by the **spacing** and **resolution** parameters, which respectively provide the distance in kilometers from the center of each box center, and the final resolution of each subdivided image. This impacts the API request time.

```
Total number of points generated:  70
List of number of points per line:  [10, 10, 10, 10, 10, 10, 10]
Image shape at 300 m resolution: (667, 664) pixels
```

The number of points generated by the subdivision, spacing and resolution of each bounding box.

After gathering all the individual areas, we can stitch them into a large file, exported as a .tiff for future use.



A stitched NDVI zone, made from subdivided bounding voxes.

In this visualisation, the lighter greens indicate a higher NDVI value (vegetation, forest) and the darker greens (urban areas and water bodies) represent areas with lower NDVI values.

II – CASE STUDY: AUSTRALIA WILDFIRES

In this section, we use the wildfire in Australia as the case study of our project and the date range we choose is from November 2019 to January 2020.

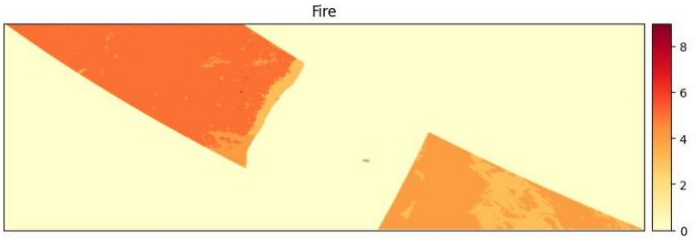
Area studied:

This case study focuses on a zone on the southern part of the east coast of Australia. The precise bounding box used is [140.9993, -37.5050, 159.1056, -28.1570].

The bounding box used for the study zone

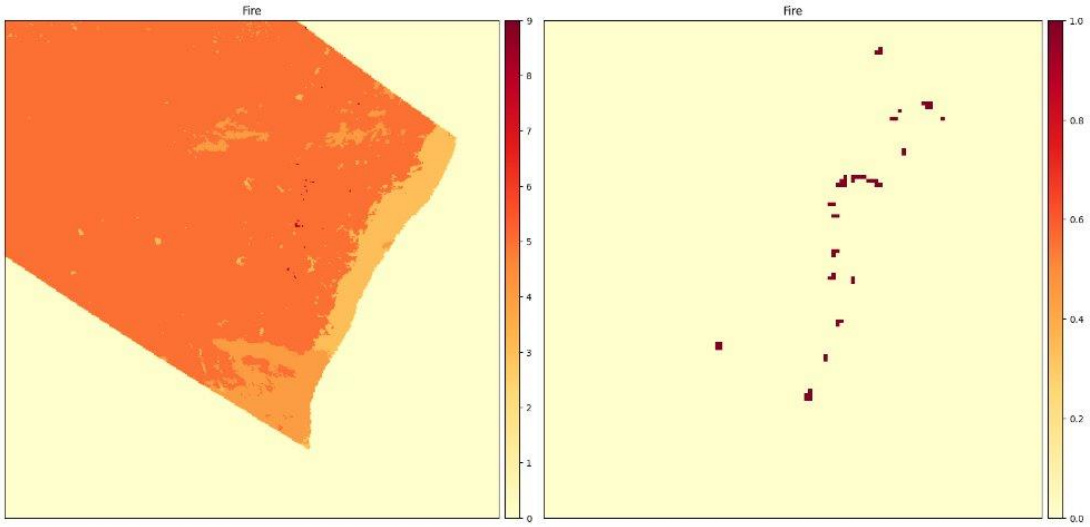
j) Modis Data

The corresponding MODIS data for that zone is the h30v12 tile.



Raw MODIS tile

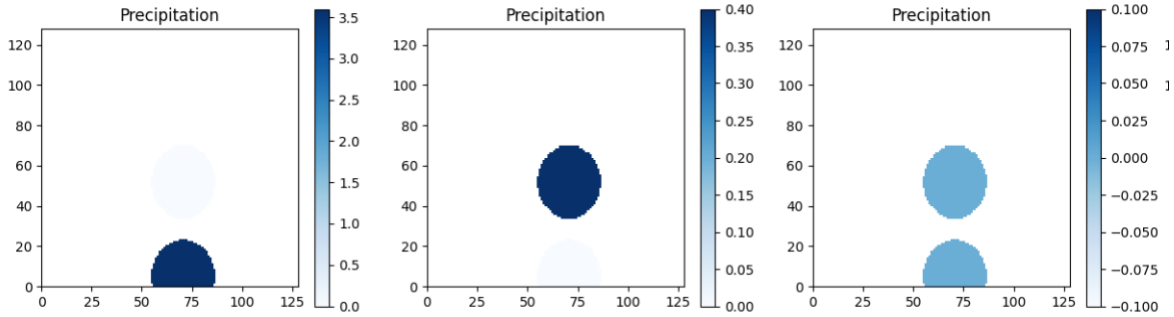
After cropping around the extracted fires, we obtain this. We isolate the fire pixels.



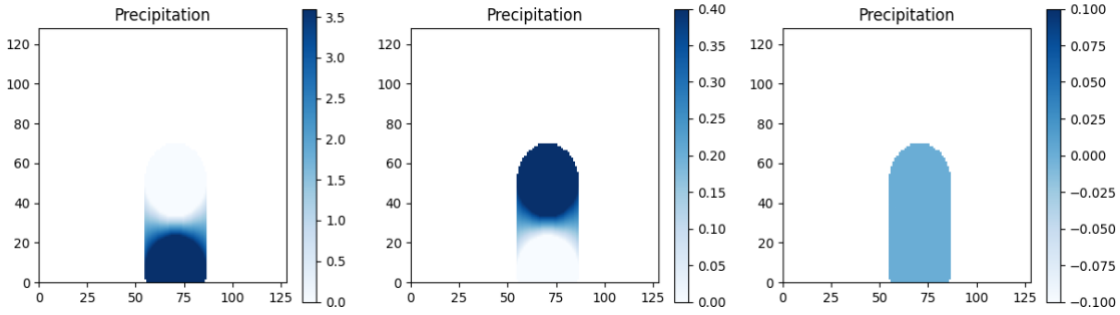
Cropped MODIS tiles. On the left is displaying the extracted fire pixels.

k) Weather Data

We gathered the weather data for the study area. One notable limitation was the lack of stations on the coast, meaning interpolating data was difficult.

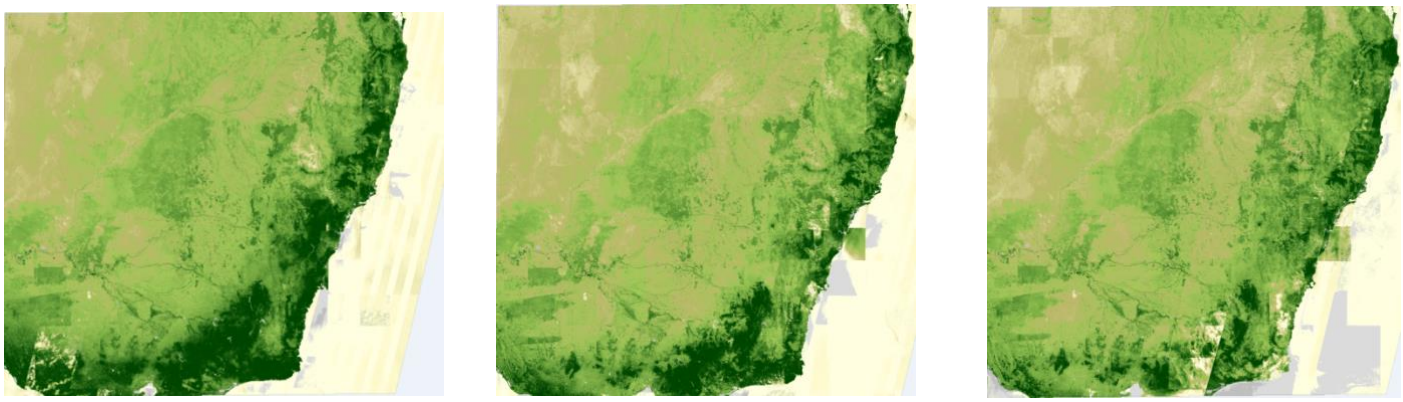


The available weather data without interpolating



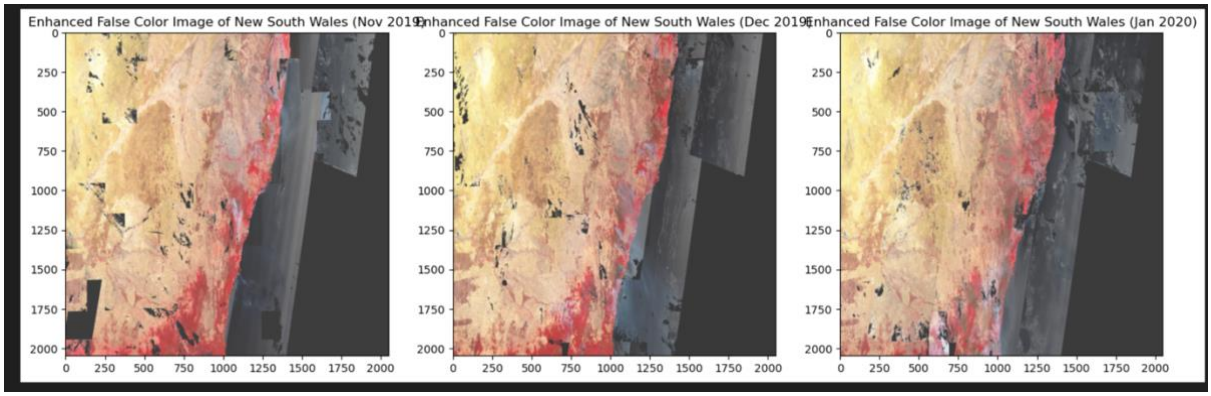
The weather data with linear interpolation using griddata

l) NDVI images



NDVI Sentinel images of Nov 2019 / Dec 2019 / Jan 2020

m) Enhanced False Color Images



False Color Images of Nov 2019 / Dec 2019 / Jan 2020

The enhanced false color image are always used to monitor the environment changes such as wildfire.

Different environmental features have different reflectance at different wavelengths: **healthy plants are more reflective** in the near-infrared bands so **they appear red** in the image. The gray and black areas represent the ocean.

These images shows clearly that the plant coverage was significantly impacted over the span of the fire event.

n) NDVI Difference Map

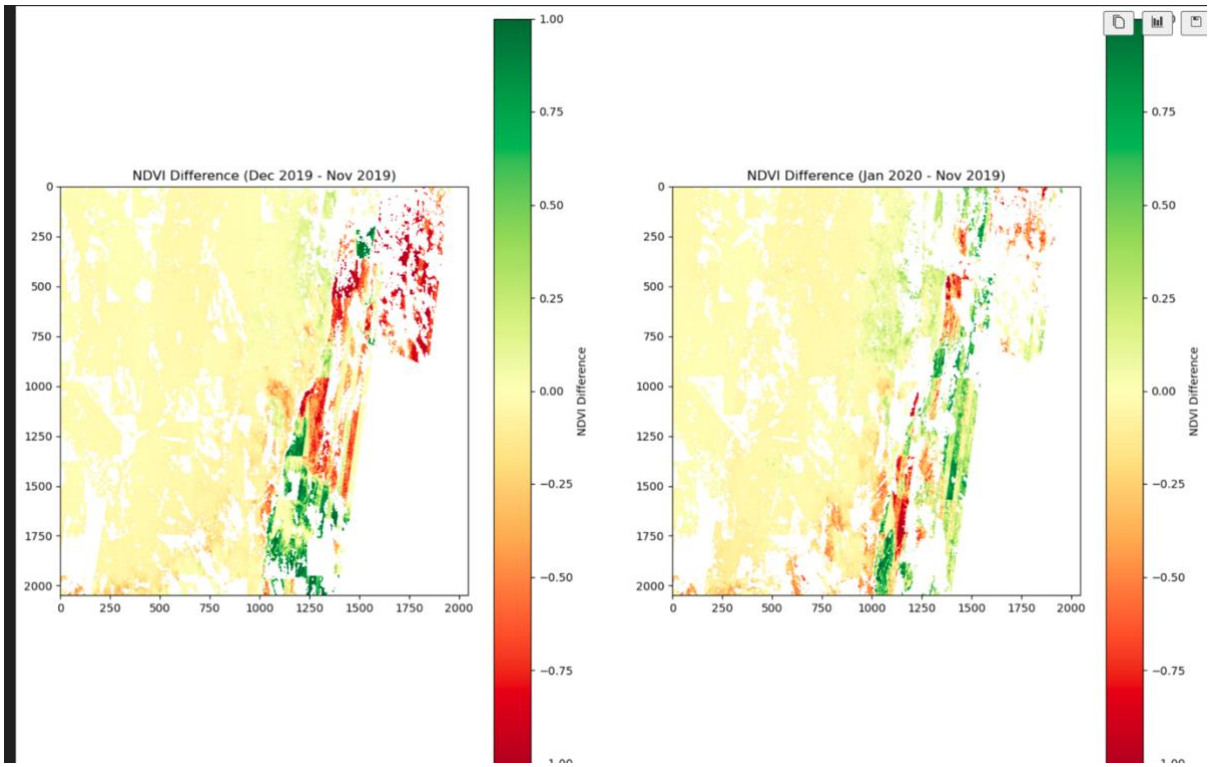
NDVI (Normalized Vegetation Index) is a common indicator used in remote sensing to monitor the density and health of vegetation.

NDVI is calculated using the formula: $\mathrm{NDVI} = \frac{\mathrm{NIR}-\mathrm{Red}}{\mathrm{NIR}+\mathrm{Red}}$

In the formula, *NIR* is the reflectance in the near-infrared band and Red is the reflectance in the red band.

<u>NDVI values range from -1 to 1:</u>

- **<u>Positive values</u>** (0 to 1) indicate **vegetation cover** (the higher the value, the denser the vegetation).
- **<u>Negative values</u>** (-1 to 0) indicate **other surface types** (e.g., water bodies, bare ground, etc.).



***<u>Red areas</u>**: Indicates reduced NDVI values and damaged or lost vegetation. These areas may be areas of vegetation loss due to fire. So the more red areas, the more reduce of plants.*

***<u>Green Area</u>**: Indicates an increase in NDVI values and recovery or growth of vegetation. So the more green areas, the more plants are recovering from the damage caused by the fire.*

<u>Difference between November 2019 and December 2019</u>

In the first map, there are significant red areas which means that there is a huge loss of plants in this period.

Difference between November 2019 and January 2020

In the second map, there are more green areas which means that plants are recovering from the wildfire.

Limitations observed

Throughout the project, we encountered multiple roadblocks that we had to address.

- GDAL

GDAL is the underlying software that most of the Python processing libraries were using for opening and rasterizing satellite images. One of the main problems we encountered was getting the support for HDF4 files working. This file format being out-of-date, support was removed from the default GDAL installation.

Reading the MODIS files required to gather extra-packages and reinstalling the Python environment to properly fix it.

- Breaking changes official API

The other main problem encountered early on was using the official Copernicus API, to get Sentinel data directly from the public servers. However, the existing community libraries available for this purpose were not working. As it turns out, changes made to the Copernicus servers in March 2023 broke existing implementations, with fixes remaining to be made by the contributors.

We had to instead opt for the private Sentinel Hub option to gather that data, though its production was done through public funding. This should be one of the main changes to be made in case of future development.

- Temporal resolution Sentinel-2

Sentinel-2 only provides a temporal resolution of 5 days, though all our other data sources provided a better 24h resolution. Sentinel being one of the main focus, we had to adjust for it by standardizing the other data into groups of 5 days to align them with the Sentinel range.

- Resolution limitations API

We addressed this last limitation by subdividing our studied zones into smaller areas, to provide the API individual lower resolution requests. We then stitched those images into a high-resolution final image.

Future improvements

Though the current status of the project is more than satisfactory, we noted down some notes to direct any future development into the project.

The first is an existing bug around the gathering of MODIS data: the current way of checking if MODIS files are available is done by checking for an available listfile. When

changing dates, the MODIS script will retrieve the wrong cache instead of fetching new data.

Another area to focus on could be to produce predictions with the processed data by our pipeline. Given the time and resource restrictions, we couldn't produce any predictions, but this could be an interesting area to target for further development.

As a proof of concept, our project focuses on retrieving data made for fire labeling given the MODIS product being retrieved. Fortunately, other types of events could be easily supported as MODIS provides other processed products, we can plug into the project to use as label data. A selection of useful products could be implemented to improve this experience.

As it stands, the project operates solely on Jupyter notebooks and Python classes. Providing a UI could be an interesting task to tackle to improve the user experience.

Conclusion

We successfully devised an easy way to gather key data necessary for the analysis of satellite images in data science, from obtaining the studied satellite data with multiple wavelengths to gathering feature data like temperature or precipitation and label data using MODIS products.

Our processing pipeline gives a streamlined way to directly combine that data into a tensor for input in a model or exporting it into standardized TIF files for further use.

This project allowed us to acquire new skills and insights into satellite image analysis, from understanding how those images are formatted and processed by the instruments, to the expectations of the scientists using them to provide a useful and desirable pipeline.