Using ‘Copernicus Data Space Ecosystem’ API Wrapper

Contents

Compiled on 2024-01-10 14:43:35.

Introduction

The CDSE package for R was developed to allow access to the 'Copernicus Data Space Ecosystem' data and services from R. The 'Copernicus Data Space Ecosystem', deployed in 2023, offers access to the EO data collection from the Copernicus missions, with discovery and download capabilities and numerous data processing tools. In particular, the 'Sentinel Hub' API provides access to the multi-spectral and multi-temporal big data satellite imagery service, capable of fully automated, real-time processing and distribution of remote sensing data and related EO products. Users can use APIs to retrieve satellite data over their AOI and specific time range from full archives in a matter of seconds. When working on the application of EO where the area of interest is relatively small compared to the image tiles distributed by Copernicus (100 x 100 km), it allows to retrieve just the portion of the image of interest rather than downloading the huge tile image file and processing it locally. The goal of the CDSE package is to provide easy access to this functionality from R.

The main functions allow to search the catalog of available imagery from the Sentinel-1, Sentinel-2, Sentinel-3, and Sentinel-5 missions, and to process and download the images of an area of interest and a time range in various formats. Other functions might be added in subsequent releases of the package.

API authentication

Most of the API functions require OAuth2 authentication. The recommended procedure is to obtain an authentication client object from the `GetOAuthClient` function, and to pass it as the `client` argument to the functions requiring the authentication. For more detailed information, you are invited to consult the "`Before you start`" document.

```
id <- Sys.getenv("CDSE_ID")
secret <- Sys.getenv("CDSE_SECRET")
OAuthClient <- GetOAuthClient(id = id, secret = secret)
```

Note

In this document, the data.frames are output as tibbles since it renders better in PDF. However, all the functions produce standard data.frames.

Collections

We can get the list of all the imagery collections available in the 'Copernicus Data Space Ecosystem'. By default, the list is formatted as a data.frame listing the main collection features. It is also possible to obtain the raw list with all information by setting the argument `as_data_frame` to `FALSE`.

```
collections <- GetCollections(as_data_frame = TRUE)
collections
#> # A tibble: 6 x 12
#>   id       title description since instrument   gsd bands constellation long.min
#>   <chr>    <chr> <chr>       <chr> <chr>      <int> <int> <chr>            <dbl>
#> 1 sentine~ Sent~ Sentinel 2~ 2015~ msi           10    13 sentinel-2        -180
#> 2 sentine~ Sent~ Sentinel 3~ 2016~ olci         300    21 <NA>              -180
#> 3 sentine~ Sent~ Sentinel 3~ 2016~ slstr       1000    11 <NA>              -180
#> 4 sentine~ Sent~ Sentinel 1~ 2014~ c-sar         NA    NA sentinel-1        -180
#> 5 sentine~ Sent~ Sentinel 2~ 2016~ msi           10    12 sentinel-2        -180
#> 6 sentine~ Sent~ Sentinel 5~ 2018~ tropomi     7000    NA <NA>              -180
#> # i 3 more variables: lat.min <dbl>, long.max <dbl>, lat.max <dbl>
```

Catalog search

The imagery catalog can be searched by spatial and temporal extent for every collection present in the `'Copernicus Data Space Ecosystem'`. For the spatial filter, you can provide either a `sf` or `sfc` object from the `sf` package, typically a (multi)polygon, describing the Area of Interest, or a numeric vector of four elements describing the bounding box of interest. For the temporal filter, you must specify the time range by either `Date` or `character` values that can be converted to date by `as.Date` function. Open intervals (one side only) can be obtained by providing the `NA` or `NULL` value for the corresponding argument.

```
dsn <- system.file("extdata", "luxembourg.geojson", package = "CDSE")
aoi <- sf::read_sf(dsn, as_tibble = FALSE)
images <- SearchCatalog(aoi = aoi, from = "2023-07-01", to = "2023-07-31",
                        collection = "sentinel-2-l2a", with_geometry = TRUE,
                        client = OAuthClient)
images
#> # A tibble: 70 x 12
#>    acquisitionDate tileCloudCover areaCoverage satellite  acquisitionTimestamp~1
#>    <date>                   <dbl>        <dbl> <chr>      <dttm>
#>  1 2023-07-31                98.9         1.84 sentinel-~ 2023-07-31 10:47:29
#>  2 2023-07-31                99.8        20.3  sentinel-~ 2023-07-31 10:47:25
#>  3 2023-07-31                99.7         5.93 sentinel-~ 2023-07-31 10:47:23
#>  4 2023-07-31                99.9        16.3  sentinel-~ 2023-07-31 10:47:14
#>  5 2023-07-31                99.9        92.5  sentinel-~ 2023-07-31 10:47:11
#>  6 2023-07-31                99.4        22.2  sentinel-~ 2023-07-31 10:47:09
#>  7 2023-07-28               100.          4.99 sentinel-~ 2023-07-28 10:37:28
#>  8 2023-07-28               100.          5.66 sentinel-~ 2023-07-28 10:37:27
#>  9 2023-07-28               100.          4.29 sentinel-~ 2023-07-28 10:37:21
#> 10 2023-07-28               100           6.85 sentinel-~ 2023-07-28 10:37:20
#> # i 60 more rows
#> # i abbreviated name: 1: acquisitionTimestampUTC
#> # i 7 more variables: acquisitionTimestampLocal <dttm>, sourceId <chr>,
#> #   long.min <dbl>, lat.min <dbl>, long.max <dbl>, lat.max <dbl>,
#> #   geometry <POLYGON [°]>
```

We can visualize the coverage of the area of interest by the satellite image tiles by plotting the footprints of the available images and showing the region of interest in red.

```
library(maps)
days <- range(as.Date(images$acquisitionDate))
maps::map(database = "world", col = "lightgrey", fill = TRUE, mar = c(0, 0, 4, 0),
          xlim = c(3, 9), ylim = c(47.5, 51.5))
plot(sf::st_geometry(aoi), add = TRUE, col = "red", border = FALSE)
plot(sf::st_geometry(images), add = TRUE)
title(main = sprintf("AOI coverage by image tiles for period %s",
                     paste(days, collapse = " / ")), line = 1L)
```

Some tiles cover only a small fraction of the area of interest, while the others cover almost the entire area.

```
summary(images$areaCoverage)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   1.845   5.603  15.113  19.758  20.346  92.463
```

The tile number can be obtained from the image attribute `sourceId`, as explained here. We can therefore summarize the distribution of area coverage by tile number, and see which tiles provide the best coverage of the AOI.

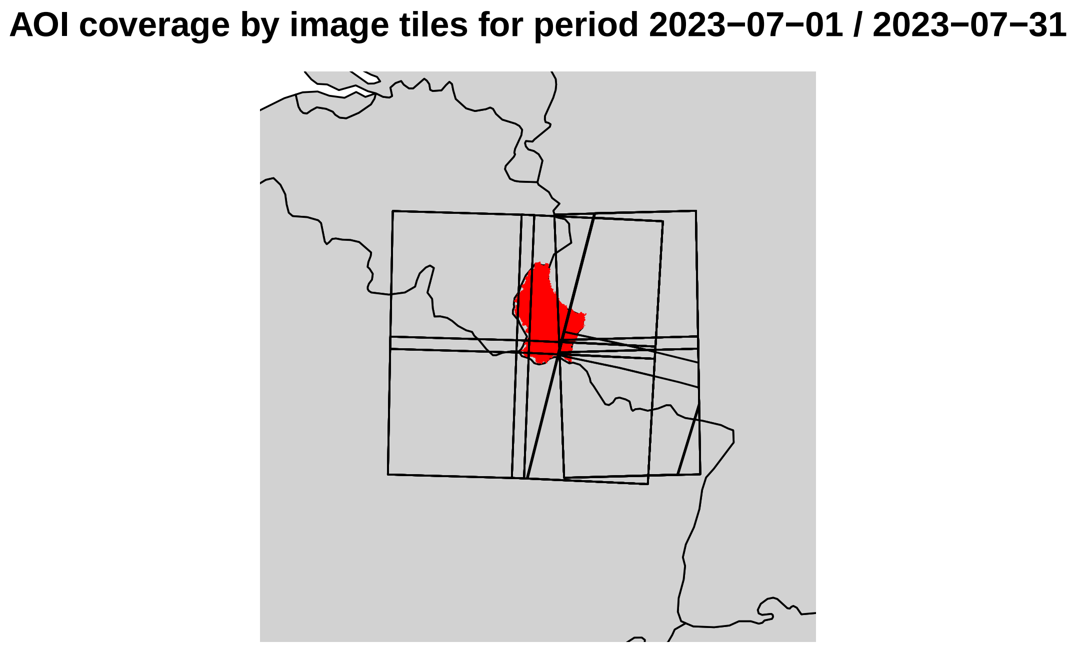

Figure 1: Luxembourg image tiles coverage

```r
tileNumber <- substring(images$sourceId, 39, 44)
by(images$areaCoverage, INDICES = tileNumber, FUN = summary)
#> tileNumber: T31UFQ
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   1.845   1.845   1.845   1.845   1.845   1.845
#> ------------------------------------------------------------
#> tileNumber: T31UFR
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   16.32   16.32   16.32   16.32   16.32   16.32
#> ------------------------------------------------------------
#> tileNumber: T31UGQ
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   4.294   4.909  12.705  12.586  20.346  20.346
#> ------------------------------------------------------------
#> tileNumber: T31UGR
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   6.855  15.608  54.299  53.426  92.463  92.463
#> ------------------------------------------------------------
#> tileNumber: T32ULA
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   6.169  14.951  18.815  17.972  22.236  22.236
#> ------------------------------------------------------------
#> tileNumber: T32ULV
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   4.944   5.603   5.820   5.723   5.934   5.934
```

Scripts

As we shall see in the examples below, we have to provide a `script` argument to the `GetArchiveImage` function.

An evalscript (or "custom script") is a piece of JavaScript code that defines how the satellite data shall be processed by the API and what values the service shall return. It is a required part of any request involving data processing, such as retrieving an image of the area of interest.

The evaluation scripts can use any JavaScript function or language structures, along with certain utility functions provided by the API for user convenience. Chrome V8 JavaScript engine is used for running the evalscripts.

The evaluation scripts are passed as the `script` argument to the `GetArchiveImage` function. It has to be either a character string containing the evaluation script or the name of the file containing the script. The `scripts` folder of this package contains a few examples of evaluation scripts.

It is beyond the scope of this document to provide guidance for writing scripts, we encourage users to consult the API Beginners Guide and Evalscript (custom script) documentation. You can find a big collection of custom scripts that you can readily use in this repository.

Retrieving AOI satellite image as a raster object

One of the most important features of the API is its ability to extract only the part of the images covering the area of interest. If the AOI is small as in the example below, this is a significant gain in efficiency (download, local processing) compared to getting the whole tile image and processing it locally.

```
dsn <- system.file("extdata", "centralpark.geojson", package = "CDSE")
aoi <- sf::read_sf(dsn, as_tibble = FALSE)
images <- SearchCatalog(aoi = aoi, from = "2021-05-01", to = "2021-05-31",
                        collection = "sentinel-2-l2a", with_geometry = TRUE,
                        client = OAuthClient)
images
#> # A tibble: 12 x 12
#>    acquisitionDate tileCloudCover areaCoverage satellite  acquisitionTimestamp~1
#>    <date>                   <dbl>        <dbl> <chr>      <dttm>
#>  1 2021-05-30              100           100.  sentinel-~ 2021-05-30 16:01:47
#>  2 2021-05-27               16.3         100.  sentinel-~ 2021-05-27 15:51:51
#>  3 2021-05-25               26.5         100.  sentinel-~ 2021-05-25 16:01:47
#>  4 2021-05-22              100           100.  sentinel-~ 2021-05-22 15:51:51
#>  5 2021-05-20               24.3         100.  sentinel-~ 2021-05-20 16:01:47
#>  6 2021-05-17                7.17        100.  sentinel-~ 2021-05-17 15:51:50
#>  7 2021-05-15               28.2         100.  sentinel-~ 2021-05-15 16:01:47
#>  8 2021-05-12                1.35        100.  sentinel-~ 2021-05-12 15:51:50
#>  9 2021-05-10               92.7         100.  sentinel-~ 2021-05-10 16:01:45
#> 10 2021-05-07               89.6         100.  sentinel-~ 2021-05-07 15:51:48
#> 11 2021-05-05              100.          100.  sentinel-~ 2021-05-05 16:01:45
#> 12 2021-05-02               78           100.  sentinel-~ 2021-05-02 15:51:48
#> # i abbreviated name: 1: acquisitionTimestampUTC
#> # i 7 more variables: acquisitionTimestampLocal <dttm>, sourceId <chr>,
#> #   long.min <dbl>, lat.min <dbl>, long.max <dbl>, lat.max <dbl>,
#> #   geometry <POLYGON [°]>
summary(images$areaCoverage)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>     100     100     100     100     100     100
```

As the area is small, it is systematically fully covered by all available images. We shall select the date with the least cloud cover, and retrieve the NDVI values as a `SpatRaster` from package `terra`. This allows further processing of the data, as shown below by replacing all negative values with zero. The size of the pixels is specified directly by the `resolution` argument. We are also adding a 100-meter `buffer` around the area of interest and `mask`ing the pixels outside of the AOI.

```
day <- images[order(images$tileCloudCover), ]$acquisitionDate[1]
script_file <- system.file("scripts", "NDVI_float32.js", package = "CDSE")
ras <- GetArchiveImage(aoi = aoi, time_range = day, script = script_file,
                       collection = "sentinel-2-l2a", format = "image/tiff",
                       mosaicking_order = "leastCC", resolution = 10,
                       mask = TRUE, buffer = 100, client = OAuthClient)
ras
#> class       : SpatRaster
#> dimensions  : 383, 355, 1  (nrow, ncol, nlyr)
#> resolution  : 0.0001003292, 0.0001003292  (x, y)
#> extent      : -73.98355, -73.94794, 40.76322, 40.80165  (xmin, xmax, ymin, ymax)
#> coord. ref. : lon/lat WGS 84 (EPSG:4326)
#> source(s)   : memory
#> name        : file444c731d66dc
```

```r
#> min value   :        -0.5069648
#> max value   :         0.9507549
ras[ras < 0] <- 0
terra::plot(ras, main = paste("Central Park NDVI on", day),
            col = colorRampPalette(c("darkred", "yellow", "darkgreen"))(99))
```

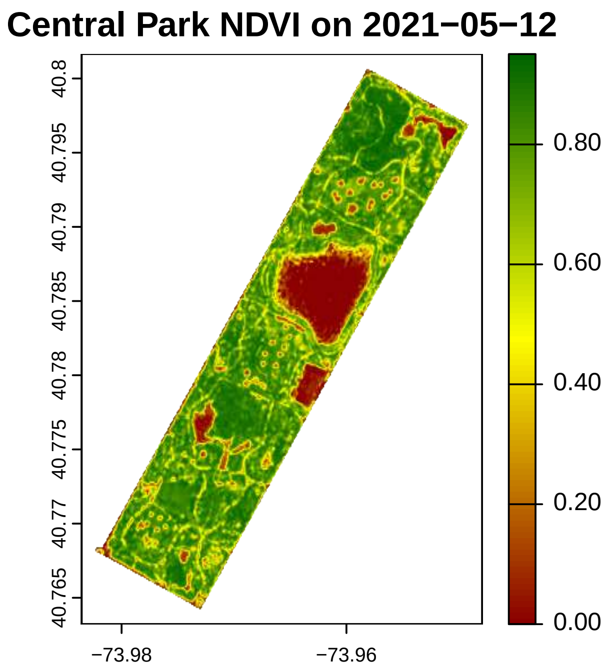

Figure 2: Central Park NDVI raster

Retrieving AOI satellite image as an image file

If we don't want to process the satellite image locally but simply use as image file (to include in a report or a Web page, for example), we can use the appropriate script that will render a three-band raster for RGB layers (or one for black-and-white image). Here we specify the area of interest by its bounding box instead of the exact geometry. We also demonstrate that the evaluation script can be passed as a single character string, and provide the number of pixels in the output image rather than the size of individual pixels - it makes more sense if the image is intended for display and not processing.

```
bbox <- as.numeric(sf::st_bbox(aoi))
script_text <- paste(readLines(system.file("scripts", "TrueColorS2L2A.js",
                                           package = "CDSE")), collapse = "\n")
cat(c(readLines(system.file("scripts", "TrueColorS2L2A.js", package = "CDSE"), n = 15), "..."), sep = "
#> //VERSION=3
#> //Optimized Sentinel-2 L2A True Color
#>
#> function setup() {
#>   return {
#>     input: ["B04", "B03", "B02", "dataMask"],
#>     output: { bands: 4 }
#>   };
#> }
#>
#> function evaluatePixel(smp) {
#>   const rgbLin = satEnh(sAdj(smp.B04), sAdj(smp.B03), sAdj(smp.B02));
#>   return [sRGB(rgbLin[0]), sRGB(rgbLin[1]), sRGB(rgbLin[2]), smp.dataMask];
#> }
#>
#> ...
png <- tempfile("img", fileext = ".png")
GetArchiveImage(bbox = bbox, time_range = day, script = script_text,
                collection = "sentinel-2-l2a", file = png, format = "image/png",
                mosaicking_order = "leastCC", pixels = c(600, 950), client = OAuthClient)
terra::plotRGB(terra::rast(png))
```

Figure 3: Central Park image as PNG file