Data Science for the Digital Atlas

Brayden Youngberg

2024-01-16

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 Introduction

This handbook is designed to provide an overview of different data types, tools, and practices to be used when working with the Adaptation Atlas ecosystem of tools and data. It can also be a useful guide for other projects as it describes the use of data on AWS S3 buckets, efficient use of cloud-optimized datasets, and notes on some best practices for data management.

2 Setting up access to the Amazon S3 bucket

To access data that is stored in private AWS S3 cloud storage buckets access credentials must be set. This guide provides step-by-step instructions for configuring AWS credentials using two methods and using them with various tools.

2.1 Method 1

The first method of setting up only needs to be configured once per device/profile. This is the suggested method if using a personal device that will access the AWS bucket often. Additional help and information can be found https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-files.html. GDAL, Python and R packages, the AWS CLI, etc. will all detect this login info automatically if it is done correctly.

Steps:

1. Create a directory called .aws in the home directory.
2. In ~/.aws/ create an empty file called 'credentials'.
    - **IMPORTANT NOTE: This is basically just a text file, but there should be there is no file extension. If the file is named credential.txt, it will not work.**
3. Open this file with a txt editor and create a [default] profile with the access_key_id and secret_access_key.
4. Create other profiles as needed with other access keys and secret keys.
5. Although not always required (GDAL presets to AWS_REGION = "us-east-1"), if dealing with buckets in multiple regions or getting an error about the region/a key not being found, a config file specifying the region and return type can be made. Keep the output set to json. *NOTE:* This is the bucket region, not the region of the user. **For the digital-atlas bucket, the region is "us-east-1"**

2.1.1 Example:

File locations:

```
Linux or macOS:
~/.aws/credentials AKA /home/USERNAME/.aws/credentials

Windows:
C:\Users\USERNAME\.aws\credentials
```

Credentials file:

```
[default]
aws_access_key_id=AKIAIOSFODNN7EXAMPLE
aws_secret_access_key=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

[read_only_user]
aws_access_key_id=AKIAI44QH8DHBEXAMPLE
aws_secret_access_key=je7MtGbClwBF/2Zp9Utk/h3yCo8nvbEXAMPLEKEY

[zarr_bucket]
aws_access_key_id=AKIAI46QZ8DHBEXAMPLE
aws_secret_access_key=xl7MjGbClwBF/2hp9Htk/h3gCo7nvbEXAMPLEKEY
```

Config file:

```
[default]
region=us-east-1
output=json

[profile read_only_user]
region=us-west-2
output=json

[profile zarr_bucket]
region=us-west-2
output=text
```

2.2 Method 2

This method sets the AWS details as environmental variables and needs to be reconfigured each time a session ends. This can also be used to override the above ~/.aws/credentials or ~/.aws/config variables if needed. It can be done from within R or Python or through the command line.

R:

```
Sys.setenv(
    AWS_ACCESS_KEY_ID = 'AKIAIOSFODNN7EXAMPLE',
    AWS_SECRET_ACCESS_KEY = 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY',
    AWS_REGION = "us-east-1"
    )
```

Python:

```
import os
os.environ['AWS_ACCESS_KEY_ID'] = 'AKIAIOSFODNN7EXAMPLE'
os.environ['AWS_SECRET_ACCESS_KEY'] = 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY'
os.environ['AWS_REGION'] = "us-east-1"
```

Linux/macOS shell:

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
export AWS_DEFAULT_REGION=us-east-1
```

3 Basic use of AWS S3 in R

This capter introduces the basics of working with the S3 bucket in R.

3.1 Using AWS with GDAL and GDAL based R tools (Terra, Stars, sf, etc.)

GDAL has special prefixes to access virtual filesystems like S3 and GCS buckets. For S3, this prefix is '/vsis3/' and more information can be found here. This prefix should be appended to any s3 file path, replacing the 's3://'.

3.1.1 Other GDAL configurations

An error may occur when accessing public buckets and datasets if you have credentials saved on your device or in your environment. This is because your credentials are being passed to a bucket where the credentials do not exist, even though the data is public. To access a public dataset without passing the aws credentials, set or pass the variable "AWS_NO_SIGN_REQUEST=[YES/NO]" to GDAL. Similarly, you may have multiple AWS profiles for different buckets or permissions saved in your .aws/credentials file (Chapter 1). If you want to use a profile other than the default, set or pass the environmental variable "AWS_PROFILE=value" where value is the name of the profile.

For the digital-atlas bucket, a GDAL virtual path this would look like:

```
/vsis3/digital-atlas/path/to/my/file.tif
```

In R terra:

```
library(terra)
cloud_cog <- rast('/vsis3/digital-atlas/MapSpam/intermediate/spam2017V2r3_SSA_V_TA-vop_tot
```

This will allow reading or writing of any file format compatible with GDAL. However, cloud-optimized formats such as COG will have the best performance.

4 Using AWS in R with S3FS

For files that are not geospatial, reading/writing large volumes of data to/from an S3 bucket, or other instances where more flexibility is needed, the S3FS R package is useful. This also provides an interface to change the access permissions of an s3 object. Multiple functions, such as upload and download have an async version which can be used with the 'future' package to do uploads in parallel. This should be used with larger uploads to save time. See the bottom of this page for an example of setting up a parallel upload.

To list buckets, folders, and files in the s3

```
#list buckets
s3_dir_ls()

#list top-level directories in the bucket
s3_dir_ls('s3://digital-atlas')

#list files in the hazard directory
s3_file_ls('s3://digital-atlas/hazards')
```

To copy a file or folder to local device from s3 bucket

```
#download a folder
s3_dir_download('s3://digital-atlas/hazards/cmip6', '~/cmip6')
#download an individual file
s3_file_download('s3://digital-atlas/boundaries/atlas-region_admin2_harmonized.gpkg',
    '~/Downloads/my-download_name.gpkg')
###Alternative option which can also be used to copy within a bucket###
#copy a folder
s3_dir_copy('s3://digital-atlas/hazards', 'hazards')
#copy an individual file
s3_file_copy('s3://digital-atlas/boundaries/atlas-region_admin2_harmonized.gpkg',
    'my-download_name.gpkg')
```

To upload a file or folder from a local device to s3 bucket

```
#upload a folder
s3_dir_upload('~/path/to/my/local/directory', 's3://mybucket/path/to/dir')
#upload an individual file
s3_file_upload('~/path/to/my/local/file.txt', 's3://mybucket/path/file.txt')
```

To move a file in the s3 bucket

```
s3_file_move('s3://mybucket/path/my_file.csv', 's3://mybucket/newpath/my_file.csv')
```

To change file or directory access permissions

```
s3_file_chmod('s3://path/to/a/file.txt', mode = 'private') # files are by default private,
s3_file_chmod('s3://path/to/a/different/file.txt', mode = 'public-read') # anyone with the
s3_file_chmod('s3://path/to/a/different/full-public/file.txt', mode = 'public-read-write')
```

More complex access permissions (i.e. specific emails, web domains, etc.) can be set in R using the paws.storage package, the AWS CLI, BOTO3/S3FS for Python, etc. This may also need to be done by setting up cross-origin resource sharing for Javascript and web applications. This can also be retrieved or set in R using a paws.storage function.

4.0.1 Use case 1 - Transfer a tif to s3 and convert it to a Cloud-optimized GeoTIFF

This is to send a single cog, but it can be easily turned into a for loop or a function for use with lapply and friends

```
# The easy way is to:
x <- rast(paste0("~/local/path/to/file.tif"))
writeRaster(x, '/vsis3/digital-atlas/path/to/where/i/want/file_cog.tif', filetype = "COG",
# However, sometimes this throws an error. If that happens, this method should work:
x <- rast(paste0("~/local/path/to/file.tif"))
writeRaster(x, paste0(tmp_d, "/cog_dir/temp_cog.tif"), filetype = "COG", overwrite = T) #c
s3_file_upload(paste0(tmp_d, "/cog_dir/temp_cog.tif"), "s3://digital-atlas/path/to/where/i
unlink(paste0(tmp_d, "/cog_dir/temp_cog.tif"))
```

4.0.2 Use case 2 - Upload a directory to the aws bucket in parallel

```
future::availableCores()
plan('multicore', workers = 5)

future <- s3_dir_upload_async(
  "~/local/path/to/productivity/data",
  's3://digital-atlas/productivity/data/',
  max_batch = fs_bytes("6MB"),
  )

upload <- value(future)
```

5 Making the most of Cloud Optimized formats

An excellent introduction to cloud-native geospatial formats can be found here. These cloud optimized formats provide the benefit of parallel and partial reading from the data source, meaning that an area of interest can be a subset from a raster file or an attribute or specific geometry can be extracted from a vector without the need to download and process the entire dataset.

This chapter is designed to provide an introduction to using some of these formats in R and Python. Conveniently, many of these formats are processed the same way their non-cloud optimized counterparts would be. It should be noted that these cloud optimized formats are still very young, and many tools and functions are still being developed for use with them. This, alongside a very active community, means that this chapter of the document will be very dynamic as new tools become available and formats become more developed.

5.1 Cloud Optimized geoTIFFs (COGs)

As the name implies, this is the cloud-optimized version of a geotiff. It uses the same (.tif) file extension, but it contains extra metadata and internal overviews. These act exactly the same as a normal geotiff and are backward compatible with standard geotiffs. To get the most from this format when using gdal-based tools, GDAL Virtual File Systems should be used to read/write data from the cloud. To do so, the following should be appended to the start of the url, depending on where the data is: - '/vsicurl/' for http/https/ftp or public aws/gcs buckets - '/vsis3/' for files in s3 buckets - '/vsigs/' for files in google cloud storage - '/vsiaz/' for files in azure cloud storage These will work for both raster and vector data and can be used in R, QGIS, GDAL CLI, etc.

```
library(terra)
# on private S3 bucket
private_cog <- rast('/vsis3/digital-atlas/MapSpam/intermediate/spam2017V2r3_SSA_V_TA-vop_t
# on public S3 bucket
Sys.setenv(AWS_NO_SIGN_REQUEST = TRUE)
public_cog <- rast(paste('"/vsis3/copernicus-dem-30m/Copernicus_DSM_COG_10_S90_00_W172_00_
```

```r
# from an https link.
https_cog <- rast('/vsicurl/https://esa-worldcover.s3.eu-central-1.amazonaws.com/v200/2021

# # same process with stars
library(stars)
private_s3_cog <- read_stars('/vsis3/digital-atlas/MapSpam/intermediate/spam2017V2r3_SSA_H
```

5.2 ZARR

This format fills the space of netCDF, HDF5 and similar multi-dimensional data cubes. Currently, the ZARR ecosystem is much more developed for Python; however, there is an active push to bring it to other languages. GDAL has some functionality for working with multi-dimensional arrays and ZARR but it is limited. This limits its use in in R spatial world for the time being. In R, Zarr datasets can be best accessed using the stars package, although with limitations and caveats.

```r
#note that currently read_mdim reads the full data set into memory,
# so anything with less than 32gb ram will likenly not work
library(stars)
Sys.setenv(AWS_NO_SIGN_REQUEST = TRUE)
zarr_store = 'ZARR:"/vsis3/cmip6-pds/CMIP6/ScenarioMIP/NOAA-GFDL/GFDL-ESM4/ssp585/r1i1p1f1
terra::sds(zarr_store)

# info = gdal_utils("mdiminfo", zarr_store, quiet = TRUE)
# jsonlite::fromJSON(info)$dimensions
# zarr <-  read_mdim(zarr_store, count = c(NA, NA, 97820))
```

The Python ecosystem is much more developed for ZARR at the time being. Xarray is the primary package for working with ZARR and other multidimensional data.

```python
import s3fs
import xarray as xr #pip install xarray[complete] for use with ZARR data

# Loading the data
s3 = s3fs.S3FileSystem(profile="ca_zarr") #a profile saved in the .aws/credentials file
def open_s3_zarr(zarr_s3path: str, s3: s3fs.core.S3FileSystem) -> xr.Dataset:
  store = s3fs.S3Map(root=zarr_s3path, s3=s3)
  return xr.open_zarr(store)

humidityZR = open_s3_zarr(
```

```
    's3://climate-action-datalake/zone=raw/source=agera5/variable=relativehumidity.zarr/',
    s3)

X = 7.65
Y = -37.33

point = humidityZR.sel(lat = X, lon = Y, method = 'nearest')
vals = point.Relative_Humidity_2m_12h.values
point.Relative_Humidity_2m_12h.plot()
```

5.3 geoparquet

This is one of the main cloud optimized formats for geospatial vector datasets. If GDAL 3.8 is installed it will use the newest geoparquet 1.0.0 spec. The same virtual file system prefixes used for rasters can be used to access these. **Before GDAL 3.5 there is no geoparquet support.** For Python users, geopandas supports the latest 1.0.0 geoparquet spec.

For R users:

```
# This section is expected to rapidly change upon release of the 'geoparquet-r package.

### Using terra (sf will be similar)
library(terra)
print(paste("GDAL version:", gdal()))
if (any(grepl("Parquet", gdal(drivers = TRUE)))) {
  print("Parquet driver available")
}

pq_uri <- '/vsis3/digital-atlas/risk_prototype/data/exposure/crop_ha_adm0_sum.parquet'
s3_parquet <- vect(pq_uri)
# or to filter it by extent while reading
aoi = ext(c(-4.5, 51.41, -34.83, -1.6))
ext_parquet <- vect(pq_uri, extent = aoi) # filter = X can be used for a vector extent
# or filter and perform operations on columns through SQL query
# NOTE: these methods can also be used for any vector format (.shp, .gpkg, etc)
sql_query <- r"(
SELECT "sum.wheat" as wheat, admin_name
FROM  crop_ha_adm0_sum
WHERE admin_name LIKE '%N%' OR "sum.wheat" > 5000
)"
```

```r
queried_parquet <- vect(pq_uri, query = sql_query)

# to write a parquet
example <- vect(ext(c(-4.5, 51.41, -34.83, -1.6)), crs = 'EPSG:4326')

x <- writeVector(example, '/vsis3/s3/path/name.parquet', filetype = 'parquet')
```

```r
### Using geoparquet-R
# Package currently in development

### Using SFarrow
# Package is depreciated and it is not encouraged unless it is the only options
# Note an error may occur if a geoparquet was made with the newest 1.0.0 spec
library(sfarrow)
# To return the full parquet in memory
sf_pq <- sfarrow::st_read_parquet("s3://digital-atlas/risk_prototype/data/exposure/crop_ha

# To access and query the parquet from out of memory first
# See the next section for more on quering with arrow datasets
library(arrow)
pq_ds <- arrow::open_dataset('s3://digital-atlas/risk_prototype/data/hazard_mean/annual/ha
filtered_sf <- pq_ds |>
    dplyr::filter(iso3 == 'UGA') |>
    sfarrow::read_sf_dataset(find_geom = T)
```

5.4 PARQUET/ARROW

This is the cloud optimized answser to tabular datasets such as csv, tsv, and excel. The Arrow package for R provides the main interface to this data format and it is very well documented. If you want to understand and take advantage of everything this format and package can offer, check out the Arrow R package documentation.

```r
### Using the r Arrow package
library(arrow)
uri <- 's3://digital-atlas/risk_prototype/data/hazard_risk_vop_ac/annual/haz_risk_vop_ac_r
#to read the full parquet in memory
parquet_tbl <- read_parquet(uri) #returns a normal dataframe of the parquet

# to access and query the parquet from out of memory
```

```r
ds <- open_dataset(uri, format = "parquet")
ds_scan <- ds$NewScan()
ds_scan$Filter(Expression$field_ref("admin0_name") == "Kenya")
filtered <- dataset <- ds_scan$Finish()$ToTable() #returns an arrow table
filtered_df <- as.data.frame(dataset)

# Arrow was also designed to play well with dplyr verbs on out-of-memory tables
library(dplyr)
ds <- read_parquet(uri, as_data_frame = FALSE) #open_dataset also works here
filtered_df <- ds |>
                filter(admin0_name == "Lesotho") |>
                collect()

#See the next section to learn how to query these using native AWS s3 methods
```

5.5 Running SQL queries on the cloud for CSV, JSON, and parquet datasets

AWS S3 allows users to query data from CSV, Parquet, or JSON files using SQL on the cloud. This allows CSV and JSON files, which are not necissarily considered "cloud-optimized", to be accessed and queried very quickly from the cloud. This can also be used for parquet files, although the returned data will always be in either csv or json format. This website is a useful help guide to the AWS S3 select SQL syntax.

```r
library(paws.storage)
bucket <- paws.storage::s3()

sql_query <- "
SELECT scenario, admin0_name
FROM S3Object
WHERE admin0_name= 'Lesotho'
"
#Note that 'S3Object', not the file path/key, is the table name in "FROM"

aws_result <- bucket$select_object_content(
    Bucket = 'digital-atlas',
    Key = 'MapSpam/raw/',
    Expression = sql_query,
    ExpressionType = 'SQL',
```

```
    InputSerialization = list(
    'CSV' = list(FileHeaderInfo = "USE")
    ),
  OutputSerialization = list(
    'CSV'= list(
      QuoteFields = "ASNEEDED"
    )
))

data <- read.csv(text = aws_result$Payload$Records$Payload, header = FALSE)
data

### Or query a parquet from S3

# A the query may have a mix of quotes in it, the r"()" raw fun. can be useful
# In this case, 'value' is both a reserved s3 select word and a column name.
# without "" surrounding value it will not be interpreted as a column
# Some complex queries allowing aggregation are also allowed
sql_query <- r"(
SELECT SUM("value") AS total_vop, AVG("value") AS avg_vop
FROM S3Object
WHERE exposure = 'vop' AND crop = 'wheat'
)"

output <- bucket$select_object_content(
    Bucket = 'digital-atlas',
    Key = 'risk_prototype/data/exposure/exposure_adm_sum.parquet',
    Expression = sql_query,
    ExpressionType = 'SQL',
    InputSerialization = list(
    'Parquet' = list()
    ),
  OutputSerialization = list(
    'CSV'= list(
      QuoteFields = "ASNEEDED"
    )
))

data <- read.csv(text = output$Payload$Records$Payload, header = FALSE)
data
```

Here are some example queries which could be of use:

```sql
-- Selects all columns where admin0 is Tanzania
SELECT *
FROM S3Object
WHERE admin0_name = 'Tanzania'


-- Selects first 5 rows (similar to head())
SELECT *
FROM S3Object
LIMIT 5

-- Calculates the average wheat vop in a dataset.
-- Again, note that value is in "" due to it being a reserved word and a column
SELECT AVG("value") as avg_wheat_vop, SUM(total_pop) as all_population
FROM S3Object
WHERE exposure = 'vop' AND crop = 'wheat'

-- Selects the unique crop names in a dataset
SELECT DISTINCT crop
FROM S3Object
```

6 Spatio-Temporal Asset Catalogs

There is currently no good way of making and editing STAC Metadata in R, so this will be a Python-centric tutorial for the time being. The RSTAC Package offers the ability to search and use STAC APIs and should work with static catalogs (such as being built in the below tutorial), in the near future. In the meantime, pystac offers a fairly simple way of searching, developing, and editing STAC Metadata.

6.1 Building STAC metadata with PySTAC

Load the required packages

```
import pystac
import datetime as dt
import shapely
#Other useful packages that help automate parts of this process:
# import riostac #this is a useful package
# import xarray stac
```

6.1.1 STAC catalogs

Catalogs are the simplest of the possible STAC specifications to make as it does not have to have any specific spatio-temporal extent to describe them. The example below shows how to build a very basic STAC catalog using pystac.

```
productivity_catalog = pystac.Catalog(
  id="crop_productivity_data",
  description="Modelled data of crop productivity",
  title="Crop Productivity"
  )
```

6.1.2 STAC collections

STAC collections are very similar to STAC catalogs, except they have extra metadata such as a spatio-temporal extent, keywords, custom fields, and STAC extensions which are explained in more detail in the STAC extension heading. They are a bit more difficult to build due to this extra metadata, but the general process is below.

```
# Build the temporal extent of the collection
# the strptime function takes a string time and the format it is in such as %Y-%M-%d
# STAC requires the time to be in UTC, so the replace function forces it into that format
# NOTE: that this method will default to YEAR-01-01 if the month and day aren't provided/N
time = ['2000', '2030']
start = str(dt.datetime.strptime(time[0], '%Y').replace(tzinfo=datetime.timezone.utc))
end = str(dt.datetime.strptime(time[1], '%Y').replace(tzinfo=datetime.timezone.utc))

# Build the spatial extent of the collection
# The bounds can be calculated using the riostac package, rasterio, terra, etc.
bounds = {'xmin': -180, 'ymin': -90, 'xmax': 180, 'ymax': 90}
bbox = [bounds['xmin'], bounds['ymin'], bounds['xmax'], bounds['ymax']]

# And now put the whole catalog together
productivity_paper_collection = pystac.Collection(id = "paper1_data",
    description = "data from paper by Todd",
    keywords = "productivity", "maize", "rice", "treated", "adaptation"
    license = "CC-BY-4.0",
    extent = pystac.Extent(
        spatial = pystac.SpatialExtent(bboxes=[bbox]),
        temporal = pystac.TemporalExtent(
            intervals=[
            start,
            end
            ]
        )
    )
)
```

6.1.3 STAC items

```
# In this example the bbox of the item is the same as the collection,
# but this is not always the case
bounds = {'xmin': -180, 'ymin': -90, 'xmax': 180, 'ymax': 90}
bbox = [bounds['xmin'], bounds['ymin'], bounds['xmax'], bounds['ymax']]
footprint = shapely.Polygon([
            [bounds['xmin'], bounds['ymin']],
            [bounds['xmin'], bounds['ymax']],
            [bounds['xmax'], bounds['ymax']],
            [bounds['xmax'], bounds['ymin']]
        ])

# Now we set the date of the item
# we could use 'strptime()' agin if date is a charachter, or set it like this:
data_date = dt.datetime(2020, 2, 5).replace(tzinfo=dt.timezone.utc)

item = pystac.Item(id='maize_productivity.tif',
                       geometry=footprint,
                       bbox=bbox,
                       datetime=data_date, #Can be set to `None` if multiple dates
                       #start_datetime=start_date, #Optional if multiple dates
                       #end_datetime=end_date,
                       properties={
                           'extraMetadataField': 'value',
                           'extraMetadataField2': 'value2',
                           "unit": "kg/ha",
                           "ssp": "SSP2-4.5"
                       })
```

Most often there will be more than one item in a collection. It is often useful to wrap the above code into a for loop and append each item to a list of items. ### STAC assets This is the final piece of the STAC spec. An asset holds the paths to the STAC Item, other data, derived datasets, and other useful assets. It offers a lot of flexibility, you can have a single asset for each date, a different asset for each model or crop, etc. depending on what fits best with the data.

```
asset = pystac.Asset(href="s3://bucket/maize_productivity.tif",
                     media_type=pystac.MediaType.COG)
item.add_asset("COG Image", asset)

# or for multiple
```

```
s3_uris = [
    "s3://bucket/maize_productivity_2020-1.tif",
    "s3://bucket/maize_productivity_2020-2.tif",
    "s3://bucket/maize_productivity_2020-3.tif"
    ]

for uri in s3_uris:
    asset = pystac.Asset(href=uri, media_type=pystac.MediaType.COG)
    asset_time = uri.split("_")[-1] # get the filename
    asset_name = f'maize productivity cog {asset_time}'
    item.add_asset(asset_name, asset)
```

6.1.4 STAC extensions

STAC extensions can be added to all of the STAC specifications - catalogs, collections, items, and assets. These extensions provide a more formal metadata framework for certain types/aspects of metadata (i.e. raster, projection, or data cube specific metadata).

```
proj = pystac.extensions.ProjectionExtension.ext(item, add_if_missing=True) #add to the it

proj.apply(epsg=4326)
```

6.1.5 Combining them all into a final catalog

```
# add the item to the collection
productivity_paper_collection.add_item(item)
# use .add_items() if you have multiple items in a list

# Add the collection to the catalog
productivity_catalog.add_child(productivity_paper_collection)

# Now just save it
productivity_catalog.normalize_and_save("~/stac/", catalog_type=pystac.CatalogType.SELF_CO
```

6.2 Update an existing catalog

References