Texas Data Repository file-level assessment script

Metadata

- *Version*: 0.0.4
- *Released*: 2025/07/08
- *Author(s)*: Bryan Gee (UT Libraries, University of Texas at Austin; bryan.gee@austin.utexas-.edu; ORCID: 0000-0003-4517-3290)
- *Contributor(s)*: None
- *License*: GNU GPLv3
- *README last updated*: 2025/07/08

Overview

This is the Jupyter Notebook version of the *dataverse-file-assessment.py* script. It contains the same functionality but is designed to provide a more detailed explanation of different steps and is targeted at an introductory level of familiarity with Python, APIs, and Dataverse. In theory, it should be accessible to a new institutional liaison for the Texas Data Repository (TDR) within a few weeks of their introduction to TDR as a whole.

Modules

You will need the following modules; several of these are not included in the default installation of Python and will need to be installed (e.g., through pip). If you need help getting started with how to install packages, please refer to this link: https://packaging.python.org/en/latest/tutorials/installing-packages/

```
import json
import os
import math
import pandas as pd
import requests
from datetime import datetime
from urllib.parse import urlparse, parse_qs
```

Toggles

Toggles are Boolean (TRUE/FALSE) variables that define whether part of the workflow should or shouldn't happen or in what fashion it should happen. For example, the *test* toggle will enable or disable 'Test' mode; when 'Test' is set to TRUE, it will cap how many records it retrieves in order to do a really quick run. Depending on the workflow and what you set the cap at, this could cut the runtime by as much as 99% to under a minute. This is ideal for making sure that the workflow as you have (re)configured it will work all the way through, rather than running in non-test mode and getting 80% of the way into an hour-long process, only to discover you have a code-breaking error at the end. There are four toggles in this workflow:

- **test**: as noted above, this will enable or disable 'Test' mode. The restricted retrieval caps are defined in the *config.json* file (see next section).
- **onlyMyInstitution**: this toggle will define whether to look at only your institution (e.g., 'UT Austin') or all institutions in TDR. As a note, only a superuser will be able to retrieve certain records (e.g., drafts) for all dataverses; institutional liaisons will not be able to get this information for an institution that they don't belong to.
- **versionsAPI**: the primary API endpoint that retrieves detailed metadata on individual datasets/files is the Native API. This endpoint only returns the most recent version's metadata. Therefore, if metadata has changed over time, you may not get a complete record of something like total file size (if files have been replaced/deleted) or authorship (if authors have been removed) because everything except the latest version is not retrieved. The versions API will return the same level of detail for all versions, but it is more time-intensive because of this, and in some instances, it may be known or suspected that outdated versions do not contribute significantly (e.g., a dataset that has only been versioned to add metadata like keywords, without file changes). If the toggle is set to 'TRUE,' the script will loop through this endpoint as well.
- **excludeDrafts**: related to *onlyMyInstitution*, if you want to ensure standardized results across all institutions and don't have superuser permissions, you may want to toggle this on to export versions of the final outputs that omit unpublished dataset.

```
#toggle for test environment (incomplete run, faster to complete)
test = True
#toggle to only look at your/one institution in TDR
onlyMyInstitution = True
#toggle for stage 3 retrieval
versionsAPI = True
#toggle for excluding unpublished
excludeDrafts = True
```

Dates

Both for filenames and in order to calculate how long the script takes, we define two timestamps. These can then be dynamically called through the script.

```
#setting timestamp at start of script to calculate run time
startTime = datetime.now()
#creating variable with current date for appending to filenames
todayDate = datetime.now().strftime("%Y%m%d")
```

config file

The *config.json* file, which can be variably named and in various formats (e.g., Michael uses a .env text file), provides parameters for the analysis. Externalizing the parameters helps to keep the script clean, since most parameters don't need to be updated regularly, and protects sensitive information like API keys. This file should never be distributed to anyone else; instead, share the *config-template.json* file.

```
#read in config file
with open('config.json', 'r') as file:
    config = json.load(file)
```

Because you can't "comment out" remarks in a JSON file in the same way that you can with Python, I want to provide some comments on what's in the config file below. The first example is showing you what's in the 'INSTITUTION' block, which is various parameters for a specific institution. You will want to modify these in the same syntax for your own institution.

```
my_institution = config['INSTITUTION']
print(my_institution)
```

```
{'name': 'University of Texas at Austin', 'filename': 'UT-
austin', 'uniqueIdentifier': 'Austin', 'ror': 'https://ror.org/00hj54h04',
'myInstitution': 'UT Austin'}
```

You don't need to load in sequential levels of a nested object; you can go directly to the field you want, as shown below.

```
##read in filename version of your institution's name
my_institution_filename = config['INSTITUTION']['filename']
###condition what goes in the filename based on toggle for which institution(s)
to ping
if onlyMyInstitution:
    institutionFilename = my_institution_filename
else:
    institutionFilename = "all-institutions"
##read in short-hand version of your institution's name
my_institution_shortName = config["INSTITUTION"]["myInstitution"]

print(f'String to add to filenames: {my_institution_filename}.\n')
print(f'Short hand version of institution name: {my_institution_shortName}.\n')
```

```
String to add to filenames: UT-austin.

Short hand version of institution name: UT Austin.
```

The 'VARIABLES' field is what dictates the size of the retrieval to be made. Limits on attributes like page size (how many in one call), page count (how many in a total call), and starting parameters will be defined in API documentation: https://guides.dataverse.org/en/latest/api/index.html. Each API is slightly different, so you will need to adapt code based on the API. This script only uses the Dataverse API, but if you need help with the Crossref, DataCite, Dryad, Figshare, OpenAlex, or Zenodo APIs, reach out to Bryan who has code for all of these. For this specific script, you should (be able to) leave these API parameters as is.

```
apiParams = config['VARIABLES']
print(apiParams)
```

```
{'PAGE_SIZES': {'datacite_prod': 1000, 'datacite_test': 200, 'dataverse':
200}, 'PAGE_LIMITS': {'datacite_test': 5, 'datacite_prod': 600, 'dspace_test':
3, 'dspace_prod': 100, 'tdr_test': 10, 'tdr_prod': 100}, 'PAGE_STARTS':
{'datacite': 0, 'dataverse': 0}, 'PAGE_INCREMENTS': {'dataverse': 1}}
```

Not all of the fields (either first-order or nested fields) are used for this script. Some of them pertain only to the other script in this repository (which isn't really that related to be honest). You won't need to bother with the various permutations of your institution's name, for example. Most of the config file is taken up by these dictionaries of key-value pairs or lists of strings. For example, the 'WORDS' field shown below is a series of words that are considered non-descriptive; these are used for a metadata assessment of dataset titles down the line.

```
words = config['WORDS']
print(words)
```

```
{'articles': ['a', 'the', 'thee', 'an'], 'conjunctions': ['and', 'or',
'but'], 'prepositions': ['in', 'to', 'of', 'for', 'with', 'as', 'from'],
'auxiliary_verbs': ['is', 'are', 'was', 'were'], 'possessives': ['our',
'my'], 'descriptors': ['data', 'dataset', 'dataverse', 'repository', 'code',
'codes', 'script', 'scripts', 'software', 'supplemental', 'supplementary',
'supporting', 'figure', 'table', 'figures', 'tables', 'file', 'files', 'et al.',
'raw', 'manuscript', 'article', 'journal', 'preprint', 'replication', 'readme',
'github', 'final', 'output'], 'order': ['S1', 'S2', 'S3'], 'version': ['v1',
'v1.0', 'v2', 'v2.0', 'v3', 'v3.0']}
```

Several fields are key-value pairs of mimeType file formats and friendly file formats. These are used to systematically assess datasets and files to look for things like whether there is a code file or a README file.

```
compressed = config['COMPRESSED_FORMATS']
print(compressed)
```

```
{'application/gzip': 'GZIP (compressed archive)', 'application/x-7z-
compressed': '7z (compressed archive)', 'application/x-gzip': 'GZIP (compressed
archive)', 'application/x-rar': 'RAR (compressed archive)', 'application/x-tar':
'TAR (compressed archive)', 'application/zip': 'ZIP (compressed archive)', 'TAR/
GNU Zip (.tar.gz/.tgz) (application/x-gzip)': 'TAR.GZ (compressed archive)',
'ZIP: PKZIP (application/zip)': 'ZIP (compressed archive)'}
```

Can you just tell me what I need to do with the *config.json* file?

There are only three things you need to change in this file to tailor it for another institution:

- **Add your Dataverse API token**: this is the first field in the config file and should be blank (““) in the template. In order to get your token, log into TDR, go to your name at the top right, and click the dropdown arrow - you should see an API token button. Do not share your token!
- **Update [‘INSTITUTION’][‘filename’]**: this is the string that will automatically be baked into filenames (so that you don’t have to manually go through and change many lines of code). This can be whatever you want.
- **Update [“INSTITUTION”][“myInstitution”]**: this is a controlled vocabulary that I cooked up for this workflow. Pick your institution out of the following:
  - “Baylor”
  - “Houston”
  - “HSC Fort Worth”
  - “SMU”
  - “TAMU”
  - “TAMU Galveston”
  - “TAMU International”
  - “Texas State”
  - “Texas Tech”
  - “Texas Women’s University”
  - “UT Arlington”
  - “UT Austin”
  - “UT San Antonio Health”
  - “UT Southwestern Medical”

Specifically for Texas A&M, enter “TAMU” to get records from all three TAMU campuses; there is a downstream process to make sure that it pulls from all three dataverses. This process currently includes some institutions that are no longer members of TDR.

Remaining set-up

The next few blocks handle the rest of the the setting up of the scripting process. The first step is to create some directories. This script is pretty simple, but in order to ensure that files are consistently saved and read from the same places for everyone, the script will first look for certain subdirectories in the folder where this script is contained and make them if they don’t exist. As you can see, if the ‘Test’ toggle is enabled, it will make a *test* subdirectory and change the directory to *test*. So test and non-test outputs are separated to avoid any confusion.

```
#creating directories
if test:
    if os.path.isdir("test"):
        print("test directory found - no need to recreate")
    else:
        os.mkdir("test")
```

```
        print("test directory has been created")
    os.chdir('test')
    if os.path.isdir("outputs"):
        print("test outputs directory found - no need to recreate")
    else:
        os.mkdir("outputs")
        print("test outputs directory has been created")
else:
    if os.path.isdir("outputs"):
        print("outputs directory found - no need to recreate")
    else:
        os.mkdir("outputs")
        print("outputs directory has been created")
```

```
test directory found - no need to recreate
test outputs directory found - no need to recreate
```

Utilized API endpoints

For maximum coverage, we need to query three different endpoints in the Dataverse API. A comparison of what information is retrieved from these

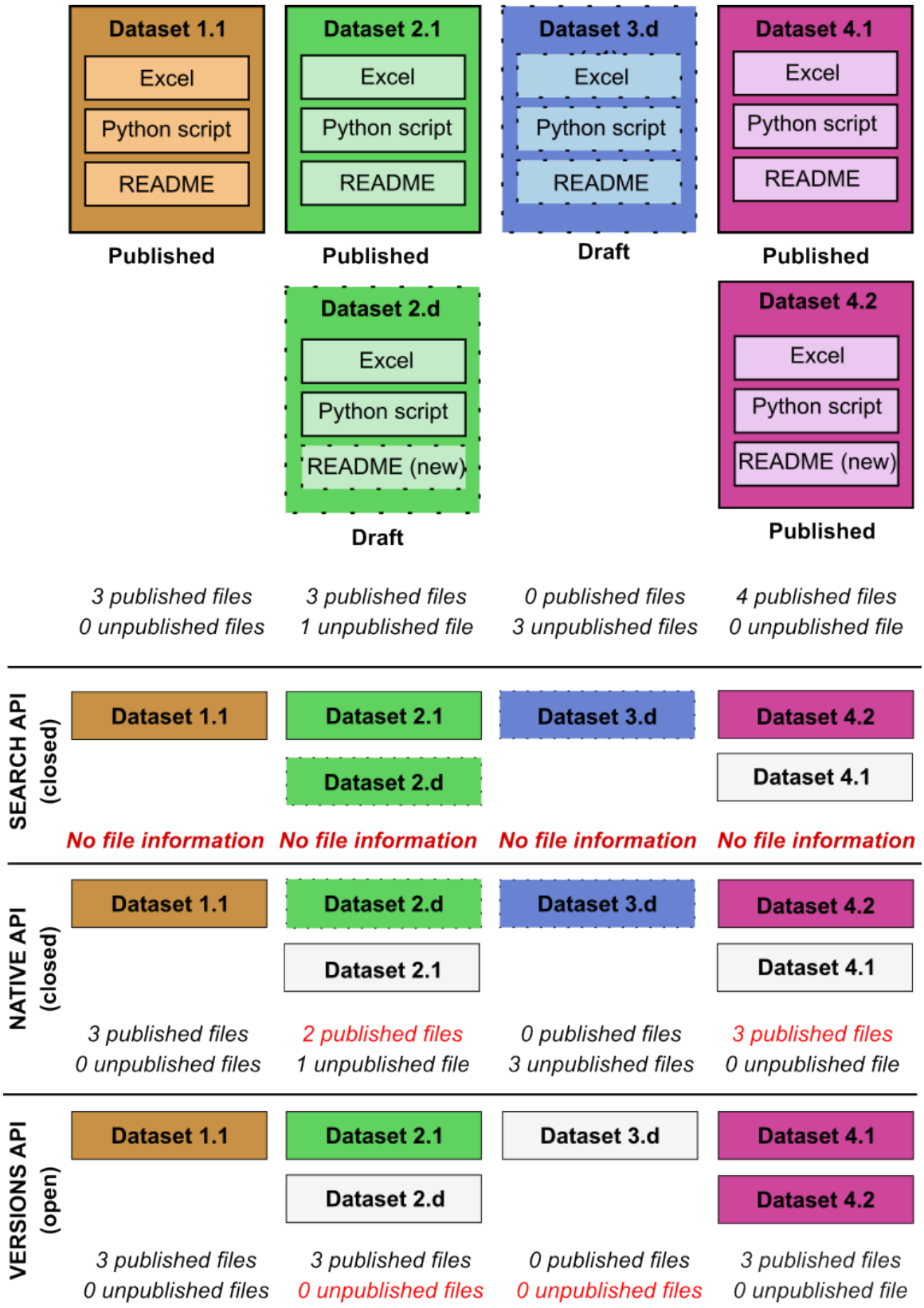

Figure 1: Comparison of information retrieved from Dataverse endpoints

Basic API params

The next step is basic API parameters. This block defines the first API endpoint (the Search API). It then dynamically calls in certain parameters about retrieval size limits from the *config.json* file, with different values based on whether you are in the Test mode. The API key is also called in. The value '*k*' is kept in here, instead of the *config.json* file, because it should always be set as zero (i.e. don't touch that). It's used to count pages while the API call is being made. The *query* value is set as the DOI prefix for all TDR deposits - this is because you cannot do a blank search through the API, so you need a value that you know will return 100% of datasets.

```
print("Beginning to define API call parameters.")
url_tdr = "https://dataverse.tdl.org/api/search/"

##set API-specific params
###Dataverse
page_limit_dataverse = config['VARIABLES']['PAGE_LIMITS']['tdr_test'] if test
else config['VARIABLES']['PAGE_LIMITS']['tdr_prod']
page_size  =  config['VARIABLES']['PAGE_SIZES']['dataverse']  if  test  else
config['VARIABLES']['PAGE_SIZES']['dataverse']

print(f"Retrieving {page_size} records per page over {page_limit_dataverse}
pages.")

###for TDR, affiliation is not reliable for returning all relevant results; the
DOI prefix is used as the most generic common denominator for datasets
query = '10.18738/T8/'
page_start_dataverse = config['VARIABLES']['PAGE_STARTS']['dataverse']
page_increment = config['VARIABLES']['PAGE_INCREMENTS']['dataverse']
k = 0

headers_tdr = {
    'X-Dataverse-key': config['KEYS']['dataverseToken']
}
```

```
Beginning to define API call parameters.
Retrieving 200 records per page over 10 pages.
```

Institution-specific parameters

The next codeblock is using what's called the 'subtree'; this is a field that contains a unique code for each institution and directs to your institution's specific dataverse. Each institution has its own set of parameters, which are largely constant between them except for the subtree, defined.

```
params_tdr_ut_austin = {
    'q': query,
    'subtree': 'utexas',
    'type': 'dataset',
```

```
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
}

params_tdr_baylor = {
    'q': query,
    'subtree': 'baylor',
    'type': 'dataset',
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
}

params_tdr_smu = {
    'q': query,
    'subtree': 'smu',
    'type': 'dataset',
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
}

params_tdr_tamu = {
    'q': query,
    'subtree': 'tamu',
    'type': 'dataset',
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
}

params_tdr_txst = {
    'q': query,
    'subtree': 'txst',
    'type': 'dataset',
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
}

params_tdr_ttu = {
    'q': query,
    'subtree': 'ttu',
    'type': 'dataset',
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
```

```
}

params_tdr_houston = {
    'q': query,
    'subtree': 'uh',
    'type': 'dataset',
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
}

params_tdr_hscfw = {
    'q': query,
    'subtree': 'unthsc',
    'type': 'dataset',
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
}

params_tdr_tamug = {
    'q': query,
    'subtree': 'tamug',
    'type': 'dataset',
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
}
params_tdr_tamui = {
    'q': query,
    'subtree': 'tamiu',
    'type': 'dataset',
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
}
params_tdr_utsah = {
    'q': query,
    'subtree': 'uthscsa',
    'type': 'dataset',
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
}
params_tdr_utswm = {
    'q': query,
    'subtree': 'utswmed',
    'type': 'dataset',
```

```
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
}

params_tdr_uta = {
    'q': query,
    'subtree': 'uta',
    'type': 'dataset',
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
}

params_tdr_twu = {
    'q': query,
    'subtree': 'twu',
    'type': 'dataset',
    'start': page_start_dataverse,
    'page': page_increment,
    'per_page': page_limit_dataverse
}
```

Parameter combinations

The next codeblock is setting up the code to handle different conditions based on whether you want to look at only your institution or all TDR institutions. The *all_params* object combines all of the institution-specific parameters. The *tamu_combined_params* object combines only TAMU parameters. In theory, you can cook up any combination you want (e.g., UT system schools).

```
all_params = {
        "UT Austin": params_tdr_ut_austin,
        "Baylor": params_tdr_baylor,
        "SMU": params_tdr_smu,
        "TAMU": params_tdr_tamu,
        "Texas State": params_tdr_txst,
        "Texas Tech": params_tdr_ttu,
        "Houston": params_tdr_houston,
        "HSC Fort Worth": params_tdr_hscfw,
        "TAMU Galveston": params_tdr_tamug,
        "TAMU International": params_tdr_tamui,
        "UT San Antonio Health": params_tdr_utsah,
        "UT Southwestern Medical": params_tdr_utswm,
        "UT Arlington": params_tdr_uta,
        "Texas Women's University": params_tdr_twu
    }

tamu_combined_params = {
```

```
        "TAMU": params_tdr_tamu,
        "TAMU Galveston": params_tdr_tamug,
        "TAMU International": params_tdr_tamui
}
```

The next codebook then uses 'if-else' statements to tell the script which combination (or single-institution parameter set) to use. The script is basically saying:

- if you set the *onlyMyInstitution* toggle to TRUE, AND you set the *my_institution_shortName* to "TAMU," then I will use the combined TAMU parameter set (*tamu_combined_params*).
- if you set the *onlyMyInstitution* toggle to TRUE, BUT you set the *my_institution_shortName* to anything other than "TAMU," then I will use the single institution parameter set (e.g., *params_tdr_ut_austin* for UT Austin).
- if you set the *onlyMyInstitution* toggle to FALSE, then I will search across all institutions with *all_params*.

```
#substitute for your institution
if onlyMyInstitution:
    if my_institution_shortName == "TAMU":
        params_list = tamu_combined_params
    else:
        params_list = {
            my_institution_shortName: all_params[my_institution_shortName]
        }
else:
    params_list = all_params
```

Functions

Functions are used when you need to apply the same logic many times throughout a script; it can make the process more concise rather than having to repeat tens to hundreds of lines of code. To be honest, the functions below are pretty technical and are the result of a lot of experimentation with many APIs. You should not touch them unless you know what you are doing, but feel free to ask questions if you want to know what a given line is doing. As a note, even though some functions' names might suggest they are designed only for querying all institutions (e.g., *retrieve_all_data_for_institutions*), if you are only running this for a single institution or a subset of institutions, it will still work the same.

```
#define functions
##function to get single page from Dataverse API
def retrieve_page_dataverse(url, params=None, headers=None):
    try:
        response = requests.get(url, params=params, headers=headers)
        response.raise_for_status()
        return response.json()
    except requests.RequestException as e:
```

```
        print(f"Error retrieving page: {e}")
        return {'data': {'items': [], 'total_count': 0}}
##function to get many pages from Dataverse API
def retrieve_all_data_dataverse(url, params, headers):
    global page_limit_dataverse, k
    # create empty list
    all_data_tdr = []

    while True and k < page_limit_dataverse:
        k+=1
        data = retrieve_page_dataverse(url, params, headers)
        total_count = data['data']['total_count']
        total_pages = math.ceil(total_count/page_limit_dataverse)
        print(f"Retrieving Page {params['page']} of {total_pages} pages...\n")

        if not data['data']:
            print("No data found.")
            break

        all_data_tdr.extend(data['data']['items'])

        #update pagination
        params['start'] += page_limit_dataverse
        params['page'] += 1

        if params['start'] >= total_count:
            print("End of response.")
            break

    return all_data_tdr

##function to retrieve many pages from many institutions
def retrieve_all_data_for_institutions(url, params_list, headers):
    all_data = []

    for institution_name, params in params_list.items():
        global page_limit_dataverse, k
        k = 0  #reset k for each institution

        all_data_tdr = retrieve_all_data_dataverse(url, params, headers)
        for entry in all_data_tdr:
            entry['institution'] = institution_name
            all_data.append(entry)

    return all_data


##function to count descriptive words
```

```
def count_words(text):
    words = text.split()
    total_words = len(words)
        descriptive_count = sum(1 for word in words if word not in
nondescriptive_words)
    return total_words, descriptive_count

##function to account for when a single word may or may not be descriptive but
is certainly uninformative if in a certain combination
def adjust_descriptive_count(row):
    if ('supplemental material' in row['titleReformatted'].lower() or
            'supplementary material' in row['titleReformatted'].lower() or
            'supplementary materials' in row['titleReformatted'].lower() or
            'supplemental materials' in row['titleReformatted'].lower()):
        return max(0, row['descriptiveWordCount_title'] - 1)
    return row['descriptiveWordCount_title']

##function to assign size bins (file or dataset level)
def assign_size_bins(df, column='fileSize', new_column='fileSizeBin'):
    df=df.copy()
    bins = [
        (0, 1 * 1024, "0-10 kB"),
        (1 * 1024, 1 * 1024 * 1024, "10 kB-1 MB"),
        (1 * 1024 * 1024, 100 * 1024 * 1024, "1-100 MB"),
        (100 * 1024 * 1024, 1 * 1024 * 1024 * 1024, "100 MB-1 GB"),
        (1 * 1024 * 1024 * 1024, 10 * 1024 * 1024 * 1024, "1-10 GB"),
        (10 * 1024 * 1024 * 1024, 15 * 1024 * 1024 * 1024, "10-15 GB"),
        (15 * 1024 * 1024 * 1024, 20 * 1024 * 1024 * 1024, "15-20 GB"),
        (20 * 1024 * 1024 * 1024, 25 * 1024 * 1024 * 1024, "20-25 GB"),
        (25 * 1024 * 1024 * 1024, 30 * 1024 * 1024 * 1024, "25-30 GB"),
        (30 * 1024 * 1024 * 1024, 40 * 1024 * 1024 * 1024, "30-40 GB"),
        (40 * 1024 * 1024 * 1024, 50 * 1024 * 1024 * 1024, "40-50 GB"),
    ]

    #default set to empty
    df[new_column] = "Empty"
    for lower, upper, label in bins:
       df.loc[(df[column] > lower) & (df[column] <= upper), new_column] = label
    #maximum bin
    df.loc[df[column] > 50 * 1024 * 1024 * 1024, new_column] = ">50 GB"

    return df
```

Retrieval process: part 1

Because we defined a series of functions related to retrieving records from the Dataverse API, potentially across all institutions, the process to trigger the API call is a single line, as seen below.

```
print("Starting TDR retrieval.\n")
all_data     =     retrieve_all_data_for_institutions(url_tdr,     params_list,
headers_tdr)
```

```
Starting TDR retrieval.

Retrieving Page 1 of 152 pages...

Retrieving Page 2 of 152 pages...

Retrieving Page 3 of 152 pages...

Retrieving Page 4 of 152 pages...

Retrieving Page 5 of 152 pages...

Retrieving Page 6 of 152 pages...

Retrieving Page 7 of 152 pages...

Retrieving Page 8 of 152 pages...

Retrieving Page 9 of 152 pages...

Retrieving Page 10 of 152 pages...
```

Subsetting API response

After the call, the API response is cached in the system but not saved in any form. You could, if you wanted to, export it as a JSON file, but I prefer working in CSV / tabular format for a lot of this work, so we'll convert it to a dataframe first. The first step in that is selecting certain fields that we want; not all fields are useful.

Why not just explode each JSON field into a separate Excel column (e.g., *authors* becomes *authors_1, authors_2,* etc.)?

I actually used to do this when I was developing another Python workflow (https://github.com/utlibraries/research-data-discovery). What I discovered is that, given any appreciable number of dataset records (let's say more than 250), at least one is virtually guaranteed to have an inordinate number of nested fields (e.g., 29 authors). This means that the script will flatten just that one field into 29 columns, which takes a lot longer to do and a lot longer to work through. If you start getting up above 1,000 records, you might not have enough RAM to actually do this. When we know what fields are important, we can just subset for them.

Why are we subsetting so few fields?

The Search API endpoint returns relatively little metadata because it's intended for a broad capture (one call, many results). It doesn't even return information like author affiliations. So the

main purpose of this call is just to get every single dataset of interest, and then we'll pass the DOIs into a different API endpoint to get more detailed metadata.

The way that this first subsetting process works is to create an empty list to hold the subsetted output (*data_select_tdr*) and then to loop through the full API response (*all_data*), pulling out specific fields. The syntax (e.g., *item.get*) is based on the hierarchy of the output. You'll see in later processes how we'll need to extract more nested objects. The script then appends each dataset's subsetted metadata into the list, and we convert it to a dataframe with the *pandas* library.

```
print("Starting TDR filtering.\n")
data_select_tdr = []
for item in all_data:
    id = item.get('global_id', '')
    type = item.get('type', '')
    institution = item.get('institution','')
    status = item.get('versionState', '')
    name = item.get('name', '')
    dataverse = item.get('name_of_dataverse', '')
    majorV = item.get('majorVersion', 0)
    minorV = item.get('minorVersion', 0)
    comboV = f"{majorV}.{minorV}"
    data_select_tdr.append({
        'institution': institution,
        'doi': id,
        'type': type,
        'status': status,
        'title': name,
        'dataverse': dataverse,
        'majorVersion': majorV,
        'minorVersion': minorV,
        'totalVersion': comboV
    })

df_data_select_tdr = pd.DataFrame(data_select_tdr)
```

```
Starting TDR filtering.
```

Post-conversion cleaning

We'll do a little bit of cleaning / modification of the data now that they're in a dataframe. The first step (filtering for 'type' = 'dataset') is redundant now because the 'type' is specified in the API call (this omits retrieval of 'dataverse' and 'file' records), but it's retained because it doesn't take up any more time and can be useful if you change the parameters later. The two other steps edit the DOI field to remove a 'doi:' string that always precedes the DOI value in that column and to add a Boolean column indicating whether a dataset has been versioned or not.

```
#remove dataverses and files
filtered_tdr = df_data_select_tdr[df_data_select_tdr['type'] == 'dataset']
#editing DOI field
filtered_tdr['doi'] = filtered_tdr['doi'].str.replace('doi:', '')
#add column for versioned
filtered_tdr['versioned']  =  filtered_tdr.apply(lambda  row:  'Versioned'  if
(row['majorVersion'] > 1) or (row['minorVersion'] > 0) else 'Not versioned',
axis=1)
```

Metadata assessment: part 1

We can't really do too much metadata assessment at the dataset/file level because of the limited metadata we get from the Search API. We can do one though: how descriptive are dataset titles? This pulls in a list of nondescriptive words from the *config.json* file (you can modify these if you wish) and then looks through the title to count how many other words remain. This operates on a few combinations of words as well (*adjust_descriptive_count* function) where the individual words aren't necessarily nondescriptive in isolation but are when combined (e.g., 'supplemental information'). Titles are formatted to remove underscores and hyphens, otherwise the entire string gets treated as one word.

```
#metadata assessments
##title
##assess 'descriptiveness of dataset title'
words = config['WORDS']
###add integers
numbers = list(map(str, range(1, 1000000)))
###combine all into a single set
nondescriptive_words = set(
    words['articles'] +
    words['conjunctions'] +
    words['prepositions'] +
    words['auxiliary_verbs'] +
    words['possessives'] +
    words['descriptors'] +
    words['order'] +
    words['version'] +
    numbers
)

filtered_tdr['titleReformatted'] = filtered_tdr['title'].str.replace('_', ' ')
filtered_tdr['titleReformatted'] = filtered_tdr['title'].str.replace('-', ' ')
#gets around text linked by underscores counting as 1 word
filtered_tdr['titleReformatted'] = filtered_tdr['titleReformatted'].str.lower()
filtered_tdr[['totalWordCount_title',     'descriptiveWordCount_title']]     =
filtered_tdr['titleReformatted'].apply(lambda x: pd.Series(count_words(x)))

filtered_tdr['descriptiveWordCount_title']                                  =
```

```
filtered_tdr.apply(adjust_descriptive_count, axis=1)
filtered_tdr['nondescriptiveWordCount_title']
=                    filtered_tdr['totalWordCount_title']                    -
filtered_tdr['descriptiveWordCount_title']
```

Retrieval process: part 2

The next part of the process is where we get richer metadata. In this step, we go through the Native API. This endpoint requires you to feed a single DOI into it, and then it returns very detailed, file-level metadata for the latest version of the dataset (regardless of publication status).

First, however, we need to deal with a quirk of the Search API: that it returns two records for any dataset that has been previously published and that is now in draft status. These records could be functionally identical (e.g., author accidentally puts published dataset into draft mode, strands it there without making changes) or quite significantly different. The code below de-duplicates these records, retaining only the published version, but saves a CSV file with a list of pairs of records with this status.

```
#sort on status, setting 'DRAFT' at bottom to remove this version for published
datasets that are in draft state, retain entry of 'PUBLISHED'
filtered_tdr = filtered_tdr.sort_values(by='status', ascending=False)
filtered_tdr.to_csv(f"outputs/{todayDate}_{institutionFilename}_all-
deposits.csv")
filtered_tdr_deduplicated    =    filtered_tdr.drop_duplicates(subset=['doi'],
keep="first")
filtered_tdr_deduplicated.to_csv(f"outputs/{todayDate}_{institutionFilename}
_all-deposits-deduplicated.csv")
print(f'Total datasets to be analyzed: {len(filtered_tdr_deduplicated)}.\n')

#create df of published datasets with draft version (retains both entries)
commonColumns = ['doi', 'title']
duplicates = filtered_tdr.duplicated(subset=commonColumns, keep=False)
dualStatusDatasets = filtered_tdr[duplicates]
dualStatusDatasets.to_csv(f"outputs/{todayDate}_{institutionFilename}_dual-
status-datasets.csv")
```

```
Total datasets to be analyzed: 100.
```

Now we have a list of unique DOIs that we want more metadata on. The following code makes a similar API call to the first one, but through the Native API endpoint. So we define the endpoint (it's a different URL) and then set up a 'for loop,' where we go through each DOI in the de-duplicated dataframe, get its metadata, and save that in an empty list called *results*.

```
#retrieving additional metadata for deposits by individual API call (one per
DOI)
##retrieves both published and never-published draft datasets; if a published
dataset is currently in DRAFT state, it will return the information for the DRAFT
state
print("Starting Native API call")
url_tdr_native = "https://dataverse.tdl.org/api/datasets/"

results = []
for doi in filtered_tdr_deduplicated['doi']:
    try:
                    response  =  requests.get(f'{url_tdr_native}:persistentId/?
persistentId=doi:{doi}', headers=headers_tdr, timeout=5)
        if response.status_code == 200:
            print(f"Retrieving {doi}\n")
            results.append(response.json())
        else:
                      print(f"Error  retrieving  {doi}:  {response.status_code},
{response.text}")
    except requests.exceptions.RequestException as e:
        print(f"Timeout error on DOI {doi}: {e}")

data_tdr_native = {
    'datasets': results
}
```

Then we do a similar subsetting process to what we did for the first API call. We make an empty list to store the output (*data_select_tdr_native*) and then loop through. You might notice that there are various *X.get* commands in the script below. This is based on the nesting of different fields. For example, *latest* is nested within *data*, so in order to get anything that is nested under *latest*, we need to define that one first. The only way to know how to structure this is to look at an actual API response. You'll see that we're subsetting a lot more fields and getting a lot more detail in the metadata. This code specifically creates an entry for each file listed in a dataset record, so some dataset-level metadata is duplicated across several rows.

```
print("Beginning dataframe subsetting\n")
data_select_tdr_native = []
for item in data_tdr_native['datasets']:
    data = item.get('data', '')
    datasetID = data.get('id', '')
    pubDate = data.get('publicationDate', '')
    latest = data.get('latestVersion', {})
    status = latest.get('versionState', '')
    status2 = latest.get('latestVersionPublishingState', '')
    doi = latest.get('datasetPersistentId', '')
    updateDate = latest.get('lastUpdateTime', '')
```

```
    createDate = latest.get('createTime', '')
    releaseDate = latest.get('releaseTime', '')
    license = latest.get('license', {})
    licenseName = license.get('name', None)
    terms = latest.get('termsOfUse', None)
    usage = licenseName if licenseName is not None else terms
    files = latest.get('files', [])
    citation = latest.get('metadataBlocks', {}).get('citation', {})
    fields = citation.get('fields', [])
    grantAgencies = []
    for field in fields:
        if field['typeName'] == 'grantNumber':
            for grant in field.get('value', []):
                        grant_number_agency = grant.get('grantNumberAgency',
{}).get('value', '')
                grantAgencies.append(grant_number_agency)
    total_filesize = 0
    unique_content_types = set()
    fileCount = len(files)
    for file_info in files:
        file_data = file_info['dataFile']
        unique_content_types.add(file_data['contentType'])
        file_entry = {
            'datasetID': datasetID,
            'doi': doi,
            #'status': status,
            'currentStatus': status2,
            'reuseRequirements': usage,
            #'fileCount': fileCount,
            #'unique_content_types': list(unique_content_types),
            'fileID': file_data.get('id', ''),
            'public': file_data.get('restricted', ''),
            'filename': file_data.get('filename', ''),
            'mimeType': file_data.get('contentType', ''),
            'friendlyType': file_data.get('friendlyType', ''),
            'tabular': file_data.get('tabularData', ''),
            'fileSize': file_data.get('filesize', 0),
            'storageIdentifier': file_data.get('storageIdentifier', ''),
            'creationDate': file_data.get('creationDate', ''),
            'publicationDate': file_data.get('publicationDate', '')
        }
        data_select_tdr_native.append(file_entry)
```

```
Beginning dataframe subsetting
```

We're also going to create a different subsetted output from the same API response, this one for individual authors (you can create as many subsetted outputs as you want from a single API

response). This one creates a separate entry for each author listed in a dataset record, so some dataset-level metadata is duplicated across several rows. This one has to deal with some extra complexity in how Dataverse structures entries that have ROR IDs vs. ones that don't.

```
#getting dataframe with entries for individual authors
author_entries = []
for item in data_tdr_native['datasets']:
    data = item.get('data', {})
    latest = data.get('latestVersion', {})
    doi = latest.get('datasetPersistentId', '')
    citation = latest.get('metadataBlocks', {}).get('citation', {})
    status2 = latest.get('latestVersionPublishingState', '')
    fields = citation.get('fields', [])
    for field in fields:
        if field['typeName'] == 'author':
            for author in field.get('value', []):
                name = author.get('authorName', {}).get('value', '')
                affiliation = author.get('authorAffiliation', {}).get('value',
''')
               identifier = author.get('authorIdentifier', {}).get('value', '')
                scheme = author.get('authorIdentifierScheme', {}).get('value',
''')
                        affiliation_expanded = author.get('authorAffiliation',
{}).get('expandedvalue', {}).get('termName', '')
                          identifier_expanded = author.get('authorIdentifier',
{}).get('expandedvalue', {}).get('@id', '')

                affiliationName = affiliation_expanded if affiliation_expanded
else affiliation
               affiliation_ror = affiliation if affiliation_expanded else None

                author_entry = {
                    'doi': doi,
                    'currentStatus': status2,
                    'authorName': name,
                    'authorAffiliation': affiliationName,
                    'rorID': affiliation_ror,
                    'authorIdentifier': identifier,
                    'authorIdentifierExpanded': identifier_expanded,
                    'authorIdentifierScheme': scheme
                }
                author_entries.append(author_entry)
```

Initial cleaning

We then convert these subsetted responses to dataframes, standardize how the DOIs are listed, and add a column for the file-level output to create a column with just the year the file was created (from a full date).

```
df_select_tdr_native = pd.json_normalize(data_select_tdr_native)
df_author_entries = pd.json_normalize(author_entries)
df_select_tdr_native['doi'] = df_select_tdr_native['doi'].str.replace('doi:',
'')
df_author_entries['doi'] = df_author_entries['doi'].str.replace('doi:', '')
df_select_tdr_native['creationDate']                                    =
pd.to_datetime(df_select_tdr_native['creationDate'])
df_select_tdr_native['fileCreationYear']                                =
df_select_tdr_native['creationDate'].dt.year
```

Metadata assessment: part 2

We do another metadata assessment at this point: binning files by their size (*size_bin*). This dataframe is then merged back with the dataframe from the Search API.

```
df_select_tdr_native          =          assign_size_bins(df_select_tdr_native,
column='fileSize', new_column='fileSizeBin')
df_select_concatenated          =          pd.merge(filtered_tdr_deduplicated,
df_select_tdr_native, on='doi', how="left")
df_select_concatenated_exist                                            =
df_select_concatenated.dropna(subset=['datasetID']).copy()            #removes
deaccessioned
df_select_concatenated_exist['datasetID']                               =
df_select_concatenated_exist['datasetID'].astype(int)
df_select_concatenated_exist.to_csv(f"outputs/{todayDate}
_{institutionFilename}_all-deposits-deduplicated_expanded-metadata.csv")

#subset to datasets that are less than version 2.0 (no major update, no file
additions)
df_select_concatenated_exist_majorVersion                               =
df_select_concatenated_exist[df_select_concatenated_exist['majorVersion'] > 1]
```

Retrieval process: part 3

This final API call is optional (*versionsAPI* toggle). If you set that to FALSE, this entire codeblock will be skipped. If you run it, the process and metadata are nearly identical to that of the previous Native API call; the Versions endpoint is technically part of the Native API, but it works a little differently. Firstly, this endpoint is public, so it will not return any metadata on drafts of any form (drafts of previously published, drafts of never published). Secondly, it returns metadata on all published versions of a dataset (the default Native endpoint only returns the most recent version). Whether the information of older published versions will significantly impact the results is a personal decision, and you may wish to experiment with it. The resultant output is subsetted and de-duplicated to handle predicted redundancy (e.g., a file present in versions 1 through 3 will have three entries), has the dataset size bin classification applied, and then aligns the columns to be the same as that of the previous outputs. This is done for both file-level output and author-level output.

```
#need to use Version endpoint to get info on published version of published
datasets that are currently in DRAFT status and all published versions of a
dataset with multiple PUBLISHED versions. This endpoint is public and does not
return any DRAFTs.
#remove datasets that have never been published (will not return any info for
this endpoint)
df_select_concatenated_exist_published = df_select_concatenated_exist_majorVersion[df_select_conca
#deduplicate on datasetID
df_select_concatenated_exist_published_dedup                                    =
df_select_concatenated_exist_published.drop_duplicates(subset="datasetID",
keep="first")

if versionsAPI:
    results_versions = []
    print("Beginning Version API query\n")
    for datasetID in df_select_concatenated_exist_published_dedup['datasetID']:
        try:
            response = requests.get(f'{url_tdr_native}{datasetID}/versions')
            if response.status_code == 200:
                print(f"Retrieving versions of dataset #{datasetID}")
                print()
                results_versions.append(response.json())
            else:
                                print(f"Error  retrieving  dataset  #{datasetID}:
{response.status_code}, {response.text}")
        except requests.exceptions.RequestException as e:
            print(f"Timeout error on DOI {doi}: {e}")

    data_tdr_versions = {
        'datasets': results_versions
    }
    print("Beginning dataframe subsetting\n")
    data_select_tdr_versions = []
    for dataset in data_tdr_versions['datasets']:
        data = dataset.get('data', [])
        for item in data:
            doi = item.get('datasetPersistentId', '')
            id = item.get("id", '')
            datasetid = item.get('datasetId', '')
            majorV = str(item.get('versionNumber', 0))
            minorV = str(item.get('versionMinorNumber', 0))
            status2 = latest.get('latestVersionPublishingState', '')
            comboV = f"{majorV}.{minorV}"
            status = item.get('versionState', '')
            for file in item.get('files', []):
                fileInfo = file['dataFile']
                data_select_tdr_versions.append({
                    'doi': doi,
```

```
                    'versionID': id,
                    'datasetID': datasetid,
                    #'majorVersion': majorV,
                    #'minorVersion': minorV,
                    'totalVersion': comboV,
                    'fileID': fileInfo.get('id', ''),
                    'filename': fileInfo.get('filename', ''),
                    'mimeType': fileInfo.get('contentType', ''),
                    'friendlyType': fileInfo.get('friendlyType', ''),
                    #'status': status,
                    'currentStatus': status2,
                    #'tabular': fileInfo.get('tabularData', ''),
                    'fileSize': fileInfo.get('filesize', ''),
                    'storageIdentifier': fileInfo.get('storageIdentifier', ''),
                    #'md5': fileInfo.get('md5', ''),
                    'creationDate': fileInfo.get('creationDate', ''),
                    'publicationDate': fileInfo.get('publicationDate', '')
                })
    #getting dataframe with entries for individual authors
    author_entries_versions = []
    for dataset in data_tdr_versions['datasets']:
        data = dataset.get('data', [])
        for item in data:
            doi = item.get('datasetPersistentId', '')
            id = item.get("id", '')
            status2 = item.get('latestVersionPublishingState', '')
            datasetid = item.get('datasetId', '')
            citation = item.get('metadataBlocks', {}).get('citation', {})
            fields = citation.get('fields', [])
            for field in fields:
                if field['typeName'] == 'author':
                    for author in field.get('value', []):
                        name = author.get('authorName', {}).get('value', '')
                                affiliation = author.get('authorAffiliation',
{}).get('value', '')
                      identifier = author.get('authorIdentifier', {}).get('value',
'')
                                scheme = author.get('authorIdentifierScheme',
{}).get('value', '')
                           affiliation_expanded = author.get('authorAffiliation',
{}).get('expandedvalue', {}).get('termName', '')
                             identifier_expanded = author.get('authorIdentifier',
{}).get('expandedvalue', {}).get('@id', '')

                    affiliationName = affiliation_expanded if affiliation_expanded
else affiliation
                           affiliation_ror = affiliation if affiliation_expanded
else None
```

```
                        author_entry = {
                            'doi': doi,
                            'currentStatus': status2,
                            'authorName': name,
                            'authorAffiliation': affiliationName,
                            'rorID': affiliation_ror,
                            'authorIdentifier': identifier,
                            'authorIdentifierExpanded': identifier_expanded,
                            'authorIdentifierScheme': scheme
                        }
                        author_entries_versions.append(author_entry)

    df_select_tdr_versions = pd.json_normalize(data_select_tdr_versions)
    df_author_entries_versions = pd.json_normalize(author_entries_versions)
                                        df_select_tdr_versions['doi']          =
df_select_tdr_versions['doi'].str.replace('doi:', '')
                                     df_author_entries_versions['doi']         =
df_author_entries_versions['doi'].str.replace('doi:', '')
    #removing duplicate entries for a given file that has not changed across
multiple versions
                                  df_select_tdr_versions['totalVersion']       =
df_select_tdr_versions['totalVersion'].astype(float)
                                            df_select_tdr_versions            =
df_select_tdr_versions.sort_values(by='totalVersion')
                                           df_select_tdr_versions_deduplicated
=                   df_select_tdr_versions.drop_duplicates(subset=['datasetID',
'storageIdentifier'], keep='first')

                                    df_select_tdr_versions_deduplicated       =
assign_size_bins(df_select_tdr_versions_deduplicated,       column='fileSize',
new_column='fileSizeBin')

                                df_select_versions_concatenated_released      =
pd.merge(df_select_tdr_versions_deduplicated,       filtered_tdr_deduplicated,
on='doi', how="left")

   #pruning and renaming columns in the two dataframes that collectively (should)
have all of the files (from the Native and the Version endpoints)
    df_version_pruned = df_select_versions_concatenated_released[["versionID",
"datasetID", "totalVersion_x", "filename", "fileID", "mimeType", "friendlyType",
"fileSize",     "storageIdentifier",     "creationDate",     "publicationDate",
"institution", "doi", "fileSizeBin", "title", "dataverse"]]
      df_version_pruned  =  df_version_pruned.rename(columns={'totalVersion_x':
'totalVersion',    'filename_x':    'filename',    'fileSize_x':    'fileSize',
'storageIdentifier_x':  'storageIdentifier',  'creationDate_x':  'creationDate',
'publicationDate_x':'publicationDate'})
                                     df_version_pruned['creationYear']        =
```

```
pd.to_datetime(df_version_pruned['creationDate'], format="%Y-%m-%d").dt.year
                                        df_version_pruned['publicationYear']
=      pd.to_datetime(df_version_pruned['publicationDate'],      format="%Y-%m-
%d").dt.year
```

The following block of code is run regardless of whether you queried the Versions endpoint or not. It standardizes some columns and creates some additional date columns.

```
df_native_pruned  =  df_select_concatenated_exist[["datasetID",  "totalVersion",
"filename",      "fileID",      "mimeType",      "friendlyType",      "fileSize",
"storageIdentifier",  "creationDate",  "publicationDate",  "institution",  "doi",
"fileSizeBin", "title", "dataverse"]]
df_native_pruned = df_native_pruned.copy()
df_native_pruned['creationYear']                                              =
pd.to_datetime(df_native_pruned['creationDate'],        format="%Y-%m-%dT%H:%M:
%SZ").dt.year
df_native_pruned['publicationYear']                                           =
pd.to_datetime(df_native_pruned['publicationDate'], format="%Y-%m-%d").dt.year
```

Conditional concatenation

If you ran the Versions endpoint process, you will need to combine that with the first Native API output and de-duplicate. This block of code uses an 'if-else' statement to either combine the two dataframes and then de-duplicate them (both file-level and author-level), or it just does the de-duplication process.

```
if versionsAPI:
       df_all_files_concat  =  pd.concat([df_version_pruned,  df_native_pruned],
ignore_index=True)
          df_all_files_concat   =   df_all_files_concat.rename(columns={'title':
'datasetTitle'})

    #deduplicate
   ##create fake versionID for drafts to ensure proper sorting and deduplicating
                                      df_all_files_concat['versionID']         =
df_all_files_concat['versionID'].fillna(9999999)
    df_all_files_concat = df_all_files_concat.sort_values(by='versionID')
                                                 df_all_files_concat_deduplicated
=            df_all_files_concat.drop_duplicates(subset=['storageIdentifier'],
keep='first')
    df_all_files_concat_deduplicated = df_all_files_concat_deduplicated.copy()
                           df_all_files_concat_deduplicated['versionID']      =
df_all_files_concat_deduplicated['versionID'].replace(9999999, None)
                  df_all_authors_concat     =     pd.concat([df_author_entries,
df_author_entries_versions], ignore_index=True)
                                               df_all_authors_concat_deduplicated
=      df_all_authors_concat.drop_duplicates(subset=['doi',      'authorName',
```

```
'authorAffiliation', 'currentStatus'], keep='first')
else:
    #sort on status and then total version, setting 'DRAFT' at bottom to remove
this version for published datasets that are in draft state, retain entry of
'PUBLISHED' and then to keep the earliest version
        df_native_pruned  =  df_native_pruned.sort_values(by=['currentStatus',
'totalVersion'], ascending=[False, True])
                                    df_all_files_concat_deduplicated        =
df_native_pruned.drop_duplicates(subset=['storageIdentifier'], keep='first')
                                          df_all_authors_concat_deduplicated
=        df_author_entries.drop_duplicates(subset=['doi',        'authorName',
'authorAffiliation', 'currentStatus'], keep='first')
```

Metadata assessment: part 3

Finally we have reached the part that is probably most interesting to some. A wide range of metadata assessments are made here (regardless of whether you queried the Versions endpoint). This includes:

- Whether a file includes 'readme', 'codebook', or 'dictionary' (proxy for data dictionary) in the filename
- Converting mimeTypes to friendlyFormats; the Dataverse API does have an existing field for this, which is included in the subsetted outputs, but I find that it still needs work to standardize, and I had this giant key-value list anyway from a different script that uses APIs that don't have friendlyFormat information.
- Whether a file is a software format (e.g., Python, R, C++)
- Whether a file is a compressed archive (e.g., .zip, .gzip, .tar)
- Whether a file is a Microsoft Office format

```
#metadata assessment
##readme presence
df_all_files_concat_deduplicated.loc[:,'isREADME']                          =
df_all_files_concat_deduplicated['filename'].str.contains('readme',
case=False)
df_all_files_concat_deduplicated.loc[:,'isCodebook']                        =
df_all_files_concat_deduplicated['filename'].str.contains('codebook',
case=False)
df_all_files_concat_deduplicated.loc[:,'isDataDict']                        =
df_all_files_concat_deduplicated['filename'].str.contains('dictionary',
case=False) #need to check sensitivity

##create separate friendlyFormat column
formatMap = config['FORMAT_MAP']
df_all_files_concat_deduplicated.loc[:,'friendlyFormat_manual']             =
df_all_files_concat_deduplicated['mimeType'].apply(
    lambda x: formatMap.get(x.strip(), x.strip()) if isinstance(x, str) and x !
= "no match found" else "no files"
```

```
)
##file formats
softwareFormats = set(config['SOFTWARE_FORMATS'].keys())
compressedFormats = set(config['COMPRESSED_FORMATS'].keys())
microsoftFormats = set(config['MICROSOFT_FORMATS'].keys())
# Assume softwareFormats is a set of friendly software format names
df_all_files_concat_deduplicated.loc[:,'isSoftware']                    =
df_all_files_concat_deduplicated['mimeType'].apply(
    lambda x: any(part.strip() in softwareFormats for part in x.split(';')) if
isinstance(x, str) else False
)
df_all_files_concat_deduplicated.loc[:,'isCompressed']                    =
df_all_files_concat_deduplicated['mimeType'].apply(
    lambda x: any(part.strip() in compressedFormats for part in x.split(';'))
if isinstance(x, str) else False
)
df_all_files_concat_deduplicated.loc[:,'isMSOffice']                    =
df_all_files_concat_deduplicated['mimeType'].apply(
    lambda x: any(part.strip() in microsoftFormats for part in x.split(';')) if
isinstance(x, str) else False
)

df_all_files_concat_deduplicated.to_csv(f"outputs/{todayDate}
_{institutionFilename}_all-files-deduplicated.csv")
```

Now you might want to combine all of the files for a given dataset into a single entry; this would be important for things like analyzing total deposit size (inclusive of all versions) or otherwise doing some of the metadata assessments at the dataset level. The following code block does that. It first defines the column(s) to be mathematically added together (only one in this case); all others will be combined into a semi-colon-delimited string. There is also some metadata editing to clean up entries (e.g., when a recombined dataset record contains 'False; True' for a Boolean variable, this is converted to 'TRUE'). For variables where some values could be duplicated (e.g., 100 text files, only a set [unique values] is returned)

```
sum_columns = ['fileSize']

def agg_func(column_name):
    if column_name in sum_columns:
        return 'sum'
    else:
        return lambda x: sorted(set(map(str, x)))

agg_funcs        =        {col:        agg_func(col)for        col        in
df_all_files_concat_deduplicated.columns if col != 'datasetID'}

df_tdr_all_files_combined = df_all_files_concat_deduplicated.groupby('datasetID').agg(agg_funcs).r
```

```
# Convert all list-type columns to comma-separated strings
for col in df_tdr_all_files_combined.columns:
   if df_tdr_all_files_combined[col].apply(lambda x: isinstance(x, list)).any():
                                        df_tdr_all_files_combined[col]     =
df_tdr_all_files_combined[col].apply(lambda x: '; '.join(map(str, x)))

tdr_all_datasets_deduplicated                                              =
df_tdr_all_files_combined.drop_duplicates(subset='datasetID', keep='first')
tdr_all_datasets_deduplicated_pruned                                       =
tdr_all_datasets_deduplicated[["datasetID",    "versionID",    "totalVersion",
"mimeType",  "friendlyType",  "fileSize",  "creationDate",  "publicationDate",
"institution",    "doi",    "datasetTitle",    "dataverse",    "creationYear",
"publicationYear",        "isREADME",        "isCodebook",        "isDataDict",
"friendlyFormat_manual", "isSoftware", "isCompressed", "isMSOffice"]]

#handles entries where aggregation returned a mixed 'False;True' value
def normalize_boolean_column(col):
    return col.apply(lambda x: True if isinstance(x, str) and 'true' in x.lower()
else False)
bool_columns   =   ["isREADME",   "isCodebook",   "isDataDict",   "isSoftware",
"isCompressed", "isMSOffice"]
tdr_all_datasets_deduplicated_pruned                                       =
tdr_all_datasets_deduplicated_pruned.copy()
for col in bool_columns:
                             tdr_all_datasets_deduplicated_pruned[col]       =
normalize_boolean_column(tdr_all_datasets_deduplicated_pruned[col])
tdr_all_datasets_deduplicated_pruned                                       =
tdr_all_datasets_deduplicated_pruned.rename(columns={'isREADME':
'containsREADME',      'isCodebook':      'containsCodebook',      'isDataDict':
'containsDataDict',    'isSoftware':    'containsSoftware',    'isCompressed':
'containsCompressed',     'isMSOffice':     'containsMSOffice',     'fileSize':
'datasetSize'})

#returns only the highest value for the version number
def extract_max_version(val):
    if isinstance(val, str):
        try:
            versions = [float(v.strip()) for v in val.split(';')]
            return max(versions)
        except ValueError:
            return val  # In case of unexpected format
    return val
tdr_all_datasets_deduplicated_pruned['totalVersion'] = tdr_all_datasets_deduplicated_pruned['total

#binning datasets by size
tdr_all_datasets_deduplicated_pruned                                       =
assign_size_bins(tdr_all_datasets_deduplicated_pruned,   column='datasetSize',
new_column='datasetSizeBin')
```

```
tdr_all_datasets_deduplicated_pruned.to_csv(f"outputs/{todayDate}
_{institutionFilename}_all-datasets-combined.csv")
```

Metadata summary

The script also returns two summary files that are likely to be of interest. The first summary is the amount of storage created by calendar year. If these numbers are discordant with previous estimates, there is a very good chance that this is because older versions are not being accounted for (either that files have been replaced or that files existed in the system well before their latest publication). For instance, if you're not paying attention, a versioned dataset most recently published in 2023 but that had a 10 GB file published in 2021, could be mistakenly counted for 2023. The second summary is the number of unique instances of file formats. The way that this calculation is done, it counts each file type once per dataset, so a dataset with 2,000 CSV files and a dataset with 1 CSV file are each counted as '1' for the total number of datasets with CSV files; this is done to eliminate the discipline-specific variability in repetitive file formats (e.g., simulation datasets with millions of text files).

```
#size summary
size_by_year     =     df_all_files_concat_deduplicated.groupby('creationYear')
['fileSize'].sum().reset_index()
size_by_year['fileGB'] = size_by_year['fileSize'] / 1000000000
print('Annual size summary')
print(size_by_year)
size_by_year.to_csv(f"outputs/{todayDate}_{institutionFilename}_annual-size-
summary.csv")
```

```
Annual size summary
   creationYear      fileSize     fileGB
0          2017  1.838593e+09   1.838593
1          2018  1.349090e+09   1.349090
2          2019  2.931868e+08   0.293187
3          2020  7.414284e+08   0.741428
4          2021  1.230189e+10  12.301894
5          2022  3.301655e+08   0.330166
6          2023  1.336943e+10  13.369429
7          2024  2.388476e+10  23.884762
8          2025  6.993964e+09   6.993964
```

```
#file format summary
##can substitute 'friendlyType' for 'mimeType' but will get some aggregating
into 'unknown'
unique_datasets_per_format                                                =
df_all_files_concat_deduplicated.groupby('friendlyFormat_manual')
['datasetID'].nunique()
```

```
print('Total file format summary')
print(unique_datasets_per_format)
unique_datasets_per_format.to_csv(f"outputs/{todayDate}_{institutionFilename}
_unique-format-summary.csv")
```

```
Total file format summary
friendlyFormat_manual
AVI                            1
Binary                        10
C Source                       1
CSV                           13
DVB Subtitle                   1
Fixed Field                    2
GIF                            1
GZIP (compressed archive)      1
HDF5                           3
JPEG                           4
JSON                           1
Jupyter Notebook               1
KMZ                            1
MATLAB                         1
MP4                            2
MS Excel                       1
MS PowerPoint                  1
MS Shortcut                    1
MS Word                        7
Makefile                       1
Markdown                       3
MiniSEED                       1
NetCDF                         4
PDF                           15
PNG                            6
Pascal                         1
Plain text                    13
PostScript                     1
Python                         3
R Notebook                     3
R Syntax                       4
SVG                            1
TAR (compressed archive)       1
TIFF                           9
TSV                           18
XFig                           1
ZIP (compressed archive)       4
Zipped Shapefile               2
Name: datasetID, dtype: int64
```

Metadata assessment: part 4

The last metadata assessment is for the author-level data. The script assesses the following:

- Whether an author has a ROR-linked affiliation
- Whether an author has a properly formatted ORCID (in Dataverse, this means that the ORCID is hyperlinked and that it appears in two fields)
- Whether an author has an ORCID, but it is not properly formatted because it lacks hyphens
- Whether an author has an ORCID, but it is not properly formatted because it is not in URL form
- Whether an author has an ORCID, but it is not properly formatted because it appears in only one of the two fields
- Whether an author has an ORCID, but it is malformed in some fashion (if any of the previous three are TRUE)
- Whether an author's name appears malformed (First Last rather than Last, First); this is based on whether there is a space in the name value but no comma
- Whether an author's initial appears malformed (probably middle, but could be first); this is based on whether there is a standalone letter with no period after it

```
#author assessment
##is ROR present
df_all_authors_concat_deduplicated = df_all_authors_concat_deduplicated.copy()
df_all_authors_concat_deduplicated.loc[:,                          'missingROR']
=             (df_all_authors_concat_deduplicated['rorID'].isna()             |
(df_all_authors_concat_deduplicated['rorID'] == ''))
##is any author ID system present
df_all_authors_concat_deduplicated.loc[:,       'missingAuthorScheme']       =
(df_all_authors_concat_deduplicated['authorIdentifierScheme'].isna() |
    (df_all_authors_concat_deduplicated['authorIdentifierScheme'] == ''))
##ORCID present and appropriately formatted
df_all_authors_concat_deduplicated.loc[:, 'properORCID'] = (
    df_all_authors_concat_deduplicated['authorIdentifierScheme'].str.upper() ==
'ORCID'
)                                                                          &
df_all_authors_concat_deduplicated['authorIdentifier'].str.contains('https://
orcid.org/', na=False)
##is ORCID present but malformatted (not hyperlinked)
df_all_authors_concat_deduplicated.loc[:,'malformedORCID_noHyphens'] = (
    df_all_authors_concat_deduplicated['authorIdentifierScheme'].str.upper() ==
'ORCID'
)  &  ~df_all_authors_concat_deduplicated['authorIdentifier'].str.contains('-',
na=False)
##is ORCID present but malformatted (no dashes)
df_all_authors_concat_deduplicated.loc[:,'malformedORCID_noURL'] = (
    df_all_authors_concat_deduplicated['authorIdentifierScheme'].str.upper() ==
'ORCID'
)                                                                          &
~df_all_authors_concat_deduplicated['authorIdentifier'].str.contains('https://
orcid.org/', na=False)
```

```
##is ORCID present but malformatted (single field)
df_all_authors_concat_deduplicated.loc[:,'malformedORCID_singleField'] = (
    df_all_authors_concat_deduplicated['authorIdentifierScheme'].str.upper() ==
'ORCID'
) & df_all_authors_concat_deduplicated['authorIdentifierExpanded'].isna()

df_all_authors_concat_deduplicated.loc[:, 'malformedORCID_any'] = (
    df_all_authors_concat_deduplicated['malformedORCID_noHyphens'] |
    df_all_authors_concat_deduplicated['malformedORCID_noURL'] |
    df_all_authors_concat_deduplicated['malformedORCID_singleField']
)
##malformed author name (order)
df_all_authors_concat_deduplicated.loc[:, 'malformedOrder'] = (
   df_all_authors_concat_deduplicated['authorName'].str.contains(' ', na=False)
&
           ~df_all_authors_concat_deduplicated['authorName'].str.contains(',',
na=False)
)
##malformed initial (standalone initial without period)
df_all_authors_concat_deduplicated.loc[:,         'malformedInitial']         =
df_all_authors_concat_deduplicated['authorName'].str.contains(r'\b[A-Z]\b(?!
\.)', regex=True)

df_all_authors_concat_deduplicated.loc[:, 'malformedName'] = (
    df_all_authors_concat_deduplicated['malformedOrder'] |
    df_all_authors_concat_deduplicated['malformedInitial']
)

df_all_authors_concat_deduplicated                                          =
df_all_authors_concat_deduplicated.sort_values(by='authorName')
df_all_authors_concat_deduplicated.to_csv(f'outputs/{todayDate}
_{institutionFilename}_all-authors.csv', index=False)
```

If you aren't a superuser, running this script across some/all TDR institutions will overweight the results in favour of your home institution because you won't pick up unpublished datasets for other institutions. The following code block uses the *excludeDrafts* toggle to determine whether to export versions of the final output files with unpublished datasets/files removed.

```
if excludeDrafts:
    #authors
   df_all_authors_concat_deduplicated_published = df_all_authors_concat_deduplicated[df_all_author
= 'DRAFT']
      df_all_authors_concat_deduplicated_published.to_csv(f'outputs/{todayDate}
_{institutionFilename}_all-authors-PUBLISHED.csv', index=False)

    #datasets
   tdr_all_datasets_deduplicated_pruned_published = tdr_all_datasets_deduplicated_pruned[tdr_all_d
```

```
& (tdr_all_datasets_deduplicated_pruned['publicationDate'] != '')]
    tdr_all_datasets_deduplicated_pruned_published.to_csv(f"outputs/{todayDate}
_{institutionFilename}_all-datasets-combined-PUBLISHED.csv")

    #files
   df_all_files_concat_deduplicated_published = df_all_files_concat_deduplicated[df_all_files_conc
& (df_all_files_concat_deduplicated['publicationDate'] != '')]
        df_all_files_concat_deduplicated_published.to_csv(f"outputs/{todayDate}
_{institutionFilename}_all-files-deduplicated-PUBLISHED.csv")

    #size summary
                                                size_by_year_published           =
df_all_files_concat_deduplicated_published.groupby('creationYear')
['fileSize'].sum().reset_index()
      size_by_year_published['fileGB'] =  size_by_year_published['fileSize'] /
1000000000
     size_by_year_published.to_csv(f"outputs/{todayDate}_{institutionFilename}
_annual-size-summary-PUBLISHED.csv")

    #file format summary
    ##can substitute 'friendlyType' for 'mimeType' but will get some aggregating
into 'unknown'
                                    unique_datasets_per_format_PUBLISHED        =
df_all_files_concat_deduplicated_published.groupby('friendlyFormat_manual')
['datasetID'].nunique()
              unique_datasets_per_format_PUBLISHED.to_csv(f"outputs/{todayDate}
_{institutionFilename}_unique-format-summary-PUBLISHED.csv")
```

```
print("Done.\n")
print(f"Time to run: {datetime.now() - startTime}\n")
if test:
    print("**REMINDER: THIS IS A TEST RUN, AND ANY RESULTS ARE NOT COMPLETE!**")
```

```
Done.

Time to run: 0:01:04.276373

**REMINDER: THIS IS A TEST RUN, AND ANY RESULTS ARE NOT COMPLETE!**
```