Using SI Open Access Data on AWS to cluster and search against American Art paintings

Introduction

In this notebook, we will show how to use the Smithsonian Open Access dataset on AWS, by demonstrating an end-to-end example of filtering metadata, downloading images, and then processing them to produce a cluster representation.

Using Dask to parse and filter collections metadata on AWS

In this section, we will use Dask and s3fs to process metadata files stored on AWS.

In [1]:
```
from dask.distributed import Client
import dask.bag as db
import json
from collections import Counter
import time
import numpy as np
import pandas as pd
import s3fs
from PIL import Image
import humanize
```

Using s3fs, we can list the top-level "directories" in the "smithsonian-open-access" S3 bucket. We can see that the bucket is split between metadata files and media files.

In [2]:
```
fs = s3fs.S3FileSystem(anon=True)
fs.ls('smithsonian-open-access')
```

Out[2]:
```
['smithsonian-open-access/media', 'smithsonian-open-access/metadata']
```

Taking a closer look inside the media directory, we can see that metadata files are organized by Smithsonian unit code. This is really helpful if you would like to work with files from just one unit, like in this case where we will be focusing on paintings from the Smithsonian American Art Museum (SAAM).

In [3]:
```python
metadata = fs.ls('smithsonian-open-access/metadata/edan')
metadata
```

Out[3]:
```
['smithsonian-open-access/metadata/edan/acah',
 'smithsonian-open-access/metadata/edan/acm',
 'smithsonian-open-access/metadata/edan/cfchfolklife',
 'smithsonian-open-access/metadata/edan/chndm',
 'smithsonian-open-access/metadata/edan/fbr',
 'smithsonian-open-access/metadata/edan/fs',
 'smithsonian-open-access/metadata/edan/fsa',
 'smithsonian-open-access/metadata/edan/fsg',
 'smithsonian-open-access/metadata/edan/hac',
 'smithsonian-open-access/metadata/edan/hmsg',
 'smithsonian-open-access/metadata/edan/hsfa',
 'smithsonian-open-access/metadata/edan/naa',
 'smithsonian-open-access/metadata/edan/nasm',
 'smithsonian-open-access/metadata/edan/nmaahc',
 'smithsonian-open-access/metadata/edan/nmafa',
 'smithsonian-open-access/metadata/edan/nmah',
 'smithsonian-open-access/metadata/edan/nmai',
 'smithsonian-open-access/metadata/edan/nmnhanthro',
 'smithsonian-open-access/metadata/edan/nmnhbirds',
 'smithsonian-open-access/metadata/edan/nmnhbotany',
 'smithsonian-open-access/metadata/edan/nmnheducation',
 'smithsonian-open-access/metadata/edan/nmnhento',
 'smithsonian-open-access/metadata/edan/nmnhfishes',
 'smithsonian-open-access/metadata/edan/nmnhherps',
 'smithsonian-open-access/metadata/edan/nmnhinv',
 'smithsonian-open-access/metadata/edan/nmnhmammals',
 'smithsonian-open-access/metadata/edan/nmnhminsci',
 'smithsonian-open-access/metadata/edan/nmnhpaleo',
 'smithsonian-open-access/metadata/edan/npg',
 'smithsonian-open-access/metadata/edan/npm',
 'smithsonian-open-access/metadata/edan/nzp',
 'smithsonian-open-access/metadata/edan/saam',
 'smithsonian-open-access/metadata/edan/si',
 'smithsonian-open-access/metadata/edan/sia',
 'smithsonian-open-access/metadata/edan/sil']
```

Within each unit, metadata is stored in .txt files in JSONL format. Here we see that there are 256 such files in the SAAM metadata directory, each a little over 100 kilobytes in size.

In [4]:
```
saam_metadata = fs.ls('smithsonian-open-access/metadata/edan/saam')
print(len(saam_metadata))
for metadata_file in saam_metadata[:5]:
    print(metadata_file)
    print(humanize.naturalsize(fs.du(metadata_file)))
```

```
256
smithsonian-open-access/metadata/edan/saam/00.txt
110.5 kB
smithsonian-open-access/metadata/edan/saam/01.txt
149.6 kB
smithsonian-open-access/metadata/edan/saam/02.txt
125.7 kB
smithsonian-open-access/metadata/edan/saam/03.txt
135.5 kB
smithsonian-open-access/metadata/edan/saam/04.txt
125.9 kB
```

Dask intro

We will be using the Python Dask library to process this collection of metadata files, because it excels at parallelizing workloads across large numbers of text files, and has built-in support for working with AWS S3 objects.

The first step to using Dask is to set up a "client" that will orchestrate processing across multiple workers. In this specific example, we use 1 worker and 4 threads per worker, because Binder only provide single CPU environments. If you are running this example on your own machine, feel free to crank these numbers up for better performance.

In [5]:
```
client = Client(threads_per_worker=4, n_workers=1)
client
```

Out[5]:

Client	**Cluster**
Scheduler: tcp://127.0.0.1:59328	**Workers:** 1
Dashboard: [http://127.0.0.1:8787/status (http://127.0.0.1:8787/status)](http://127.0.0.1:8787/status)	**Cores:** 4
	Memory: 17.18 GB

To process nested JSON data, we use the Dask "bag" datatype, and tell it we want to process all .txt files in the SAAM metadata directory. This command will run instantaneously, because Dask uses a "lazy" execution model. This means that most Dask commands build an execution graph to be run later.

In [6]:
```
b = db.read_text('s3://smithsonian-open-access/metadata/edan/saam/*.txt'
,
                storage_options={'anon': True}).map(json.loads)
```

One exception to the "lazy" execution model is the take command, which executes the code immediately to process the first few objects. Here we take just the first object, and then save it to file.

Take a look at how complicated these objects are: [saam_metadata_example.json (https://github.com/sidatasciencelab/siopenaccess/blob/master/saam_metadata_example.json)](https://github.com/sidatasciencelab/siopenaccess/blob/master/saam_metadata_example.json)

In [7]:
```
saam_example = b.take(1)[0]
with open('saam_metadata_example.json', 'w') as json_out:
    json.dump(saam_example, json_out, indent=2)
```

In [8]:
```
#This interactive JSON viewer only works correctly in Jupyter Lab

from IPython.display import display, JSON
display(JSON(saam_example, expanded=True))
```

```
<IPython.core.display.JSON object>
```

Because of how complicated each individual record is, we need to create a Python function to pull out the pieces of data that we want to store and/or filter on later.

In [9]:
```
def flatten(record):
    """Take a single SAAM metadata record, and pulls out specific pieces
of data.

    Parameters
    ----------
    record : dict
        A single SAAM metadata record in highly-nested dictionary forma
t.

    Returns
    -------
    flattened_record: dict
        An un-nested dictionary that only contains the record id, unit c
ode,
        object title, media_count, media_id, topic list, object type, an
d
        object medium.
    """
    flattened_record = dict()
    flattened_record['id'] = record['id']
    flattened_record['unitCode'] = record['unitCode']
    flattened_record['title'] = record['title']
    media_count = record['content'].get('descriptiveNonRepeating', {}).g
et('online_media',{}).get('mediaCount',np.nan)
    flattened_record['media_count'] = float(media_count)
    media = record['content'].get('descriptiveNonRepeating', {}).get('on
line_media',{}).get('media',[])
    if len(media):
        flattened_record['media_id'] = media[0]['idsId']

    topics = record['content'].get('indexedStructured',{}).get('topic',
[])
    if len(topics):
        flattened_record['topics'] = '|'.join(topics)

    if 'freetext' in record['content']:
        if 'objectType' in record['content']['freetext']:
            for obtype in record['content']['freetext']['objectType']:
                if obtype['label'] == 'Type':
                    flattened_record['object_type'] = obtype['content']
        if 'physicalDescription' in record['content']['freetext']:
            for phys in record['content']['freetext']['physicalDescripti
on']:
                if phys['label'] == 'Medium':
                    flattened_record['medium'] = phys['content']
        if 'name' in record['content']['freetext']:
            for name in record['content']['freetext']['name']:
                if name['label'] == 'Artist':
                    flattened_record['artist'] = name['content']
        if 'date' in record['content']['freetext']:
            for date in record['content']['freetext']['date']:
                if date['label'] == 'Date':
                    flattened_record['date'] = str(date['content'])

    return flattened_record
```

Here we test out this `flatten` function by passing it the single record we pulled out earlier with the `take` command. You can see how it converted the highly-nested format into a single level dictionary with only a few pieces of information.

In [10]:
```
flattened_example = flatten(saam_example)
flattened_example
```

Out[10]:
```
{'id': 'edanmdm-saam_1971.439.94',
 'unitCode': 'SAAM',
 'title': 'Calavera for the Policeman',
 'media_count': 1.0,
 'media_id': 'SAAM-1971.439.94_1',
 'topics': 'Occupations|Service|Policeman|Skeleton',
 'object_type': 'Graphic Arts-Print',
 'medium': 'woodcut',
 'artist': 'José Guadalupe Posada, Mexican, Aguascalientes, Mexico 1852
-died Mexico City, Mexico 1913'}
```

Finally, we send all 12,542 metadata records through the `flatten` function with the Dask `map` function, and ensure that the command is actually executed by using the `compute` function. Since `flatten` returns a single-level dictionary, we can convert the results of into a table using the `to_dataframe` function. Then we run `head` to look at the first 5 rows of this table.

In [11]:
```
saam_json = b.map(flatten).compute()
saam_df = pd.DataFrame(saam_json)
saam_df.head()
```

Out[11]:

	id	unitCode	title	media_count	media_id	
0	edanmdm-saam_1971.439.94	SAAM	Calavera for the Policeman	1.0	SAAM-1971.439.94_1	Occupations|Service
1	edanmdm-saam_1915.5.1	SAAM	The Falling Gladiator	1.0	SAAM-1915.5.1_1	Sport|Occupa
2	edanmdm-saam_1983.90.173	SAAM	The Sortie Made by the Garrison of Gibraltar i...	NaN	NaN	
3	edanmdm-saam_1985.66.295_540	SAAM	Ta-do-udo-sa (Prairie Chicken)	1.0	SAAM-1985.66.295540_1	Et
4	edanmdm-saam_1930.12.47	SAAM	Figure Study for Decorative Panel	1.0	SAAM-1930.12.47_1	Landscapes|

Taking a look at the structure of the `pandas` dataframe, we can see that some of the data fields are null.

In [12]:
```
saam_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12542 entries, 0 to 12541
Data columns (total 10 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   id            12542 non-null  object 
 1   unitCode      12542 non-null  object 
 2   title         12542 non-null  object 
 3   media_count   11561 non-null  float64
 4   media_id      11561 non-null  object 
 5   topics        11230 non-null  object 
 6   object_type   12542 non-null  object 
 7   medium        12100 non-null  object 
 8   artist        12389 non-null  object 
 9   date          10575 non-null  object 
dtypes: float64(1), object(9)
memory usage: 980.0+ KB
```

Let's look at the most common "object_type" and "medium" combinations amongst all SAAM works of art. We can see that the most common type of art is a painting created with oil on canvas.

```
In [13]: saam_df.groupby(['object_type','medium']).size().sort_values(ascending=F
alse).head(20)
```

```
Out[13]: object_type              medium
         Painting                 oil on canvas                          1662
                                  watercolor on paper                     853
         Drawing                  pencil on paper                         642
         Graphic Arts-Print       wood engraving on paper                 441
         Drawing                  drawing                                 354
         Painting-Miniature       watercolor on ivory                     350
         Photography-Photoprint   albumen silver print                    268
         Graphic Arts-Print       lithograph                              223
         Decorative Arts-Glass    glass                                   223
         Graphic Arts-Print       etching on paper                        214
         Sculpture                plaster                                 209
         Drawing                  pencil                                  201
         Graphic Arts-Print       etching                                 194
         Painting                 watercolor                              149
         Graphic Arts-Print       engraving                               149
         Painting                 oil on wood                             137
         Graphic Arts-Print       hand-colored lithograph on paper        113
                                  lithograph on paper                     112
                                  wood engraving                          111
         Painting                 watercolor and pencil on paper          110
         dtype: int64
```

Since works of art have multiple topics listed, it is slightly more complicated to look at the most common topics.

In [14]:
```
def count_topics(topic_column):
    """
    Take the '|'-concatenated column from a pandas dataframe, and expand
    it into a Counter object to see the most common individual topics.

    Parameters
    ----------
    topic_column : pandas Series
        A column from a metadata table. It is expected that multiple top
ics
        are separated with a pipe symbol.

    Returns
    -------
    topic_counts: Counter
        A Python Counter object of each unique topic, and the number of
 times
        that it is listed.
    """
    topic_list = []
    topics_entries = topic_column.dropna().tolist()
    for topics_entry in topics_entries:
        topics = topics_entry.split('|')
        if len(topics):
            topic_list += topics
    topic_counts = Counter(topic_list)
    return topic_counts
```

Out of 12,542 works of art, there are 4,016 unique topics totally 56,927 total topics listed. We can see that "Landscapes" is the most common topic.

In [15]:
```
topic_counts = count_topics(saam_df['topics'])

print(len(topic_counts))
topic_counts.most_common(20)
```

```
4016
```

Out[15]:
```
[('Landscapes', 3368),
 ('Architecture', 2482),
 ('Portraits', 2459),
 ('Figure group', 2202),
 ('Men', 1988),
 ('Occupations', 1534),
 ('Ethnicity', 1375),
 ('Animals', 1354),
 ('Figure female', 1128),
 ('Women', 1062),
 ('Clothing and dress', 1003),
 ('Figure male', 814),
 ('Botanical study', 776),
 ('Religion', 751),
 ('Nudity', 707),
 ('Domestic', 704),
 ('Cityscapes', 625),
 ('Recreation', 558),
 ('Dress accessories', 554),
 ('Children', 517)]
```

Since 11,561 images is a lot to process, let's try to filter out all art that include people.

In [16]:
```
include_topics = ['Landscapes','Architecture','Animals']
exclude_topics = ['Portraits','Nudity','Ethnicity','Men','Women','Childr
en',
                 'Figure male','Figure female','Figure group',
                  'Botanical study']
include_regex = '|'.join(include_topics)
exclude_regex = '|'.join(exclude_topics)
filtered_df = saam_df[(saam_df['topics'].str.contains(include_regex, reg
ex=True).fillna(False)) & \
                  (~saam_df['topics'].str.contains(exclude_regex, regex=T
rue).fillna(False))]
filtered_df.head()
```

Out[16]:

	id	unitCode	title	media_count	media_id	
6	edanmdm-saam_1929.6.144	SAAM	The Brook, Greenwich, Connecticut	1.0	SAAM-1929.6.144_1	Lands
20	edanmdm-saam_1991.56.271	SAAM	Nanfio	1.0	SAAM-1991.56.271_1	
21	edanmdm-saam_1998.160.3	SAAM	Untitled from "Atlantic and Great Western Rail...	NaN	NaN	Bridges|Atlantic and Grea
25	edanmdm-saam_1983.83.171	SAAM	Rhine at the Lurlei	1.0	SAAM-1983.83.171_1	Lurlei|Landscape
26	edanmdm-saam_1983.83.55	SAAM	Untitled (transfer drawing for Storm near Timb...	1.0	SAAM-1983.83.55_1	Trees|Western|Weather|Lar

After running this filter step, we are left with 1,573 unique topics from 2,816 works of art.

In [17]:
```
print(len(filtered_df))
filtered_topics = count_topics(filtered_df['topics'])
print(len(filtered_topics))
filtered_topics.most_common(20)
```

```
2816
1573
```

Out[17]:
```
[('Landscapes', 1979),
 ('Architecture', 1177),
 ('Animals', 634),
 ('Mountains', 379),
 ('Rivers', 315),
 ('Trees', 291),
 ('Cityscapes', 276),
 ('Boats and boating', 258),
 ('Domestic', 253),
 ('Waterscapes', 231),
 ('Religion', 225),
 ('Detail', 189),
 ('Dwellings', 166),
 ('Birds', 164),
 ('Coasts', 141),
 ('Time', 125),
 ('Water', 122),
 ('Weather', 119),
 ('Seasons', 119),
 ('Roads', 112)]
```

Finally, let's limit our search to only paintings. This gives us a target set of 808 images to download and process.

In [36]:
```
filtered_paintings = filtered_df[(filtered_df['object_type'] == 'Paintin
g') &
                                 (pd.notnull(filtered_df['media_id']))]
filtered_paintings.head()
```

Out[36]:

	id	unitCode	title	media_count	media_id	
6	edanmdm-saam_1929.6.144	SAAM	The Brook, Greenwich, Connecticut	1.0	SAAM-1929.6.144_1	Landsc
34	edanmdm-saam_1967.136.6	SAAM	Mountains in Colorado	1.0	SAAM-1967.136.6_1	Mountains|West
36	edanmdm-saam_1958.5.3	SAAM	Above Tower Falls, Yellowstone	1.0	SAAM-1958.5.3_1	Yellowstone National Park|L
37	edanmdm-saam_1984.50	SAAM	The Departure of the Crusaders	1.0	SAAM-1984.50_2	Bishop|Crusades|Architectur
39	edanmdm-saam_1973.150	SAAM	The Yacht America	1.0	SAAM-1973.150_1	Boats and boating|Architec

In [37]:
```
filtered_paintings.to_csv('saam_painting_metadata.tsv', index=False, sep
='\t')
```

In [38]:
```
painting_ids = filtered_paintings['media_id'].tolist()
painting_ids[:10]
```

Out[38]:
```
['SAAM-1929.6.144_1',
 'SAAM-1967.136.6_1',
 'SAAM-1958.5.3_1',
 'SAAM-1984.50_2',
 'SAAM-1973.150_1',
 'SAAM-1985.66.385_1',
 'SAAM-1909.7.51_1',
 'SAAM-1983.95.91_1',
 'SAAM-1972.2.12_1',
 'SAAM-1978.68_1']
```

Download image files from S3

This section of the demo can actually be skipped, since the GitHub repository already has all thumbnails included, and this is the slowest part of the demo (around 4 minutes).

In [21]:
```
def download_thumbnail(edan_id):
    """
    Opens a full-size image from S3, compresses it to thumbnail, and wri
tes
    it to disk. Harcoded for SAAM images, and to save to saam_thumbnails
directory.

    Parameters
    ----------
    edan_id : string
        The Smithsonian Enterprise Digital Asset Network (EDAN) ID of th
e object
        to download.

    """
    thumb_size = (500, 500)
    s3_url = f'smithsonian-open-access/media/saam/{edan_id}.jpg'
    file_dest = f'saam_thumbnails/{edan_id}.jpg'
    with fs.open(s3_url,'rb') as s3_image:
        pil_image = Image.open(s3_image)
        pil_image.thumbnail(thumb_size)
        pil_image.save(file_dest)
    return
```

Un-comment the code block below (by removing the # at the beginning of each line) if you would like to run this download step anyways.

In [22]:
```
#start = time.time()
#futures = client.map(download_thumbnail, painting_ids)
#results = client.gather(futures)
#end = time.time()
#print(end - start)
```

Producing image feature vectors with TensorFlow

Now that we have all of our painting images downloaded into the "saam_thumbnails" directory, we can do some processing on them.

For this demonstration, we will attempt to use a broad machine learning technique called "unsupervised learning" (https://en.wikipedia.org/wiki/Unsupervised_learning (https://en.wikipedia.org/wiki/Unsupervised_learning)) to cluster similar images together, and learn a little bit about our collection.

Specifically, we will be using a pre-trained photographic image classifier called MobileNetV2 (https://arxiv.org/abs/1801.04381 (https://arxiv.org/abs/1801.04381)) to extract a numerical "feature vector" representation of each image, which can then be used to calculate image-to-image distances. We will then feed those vectors into the UMAP (https://arxiv.org/abs/1802.03426 (https://arxiv.org/abs/1802.03426)) algorithm, which reduces our multi-thousand dimensional feature space down to an easy-to-interpret 2-dimensional representation.

In [23]:
```
import tensorflow as tf
from umap import UMAP
import os
from PIL import Image
from sklearn.decomposition import PCA
from sklearn.preprocessing import minmax_scale
import matplotlib.pyplot as plt
%matplotlib inline
```

First, we use the TensorFlow Keras deep learning library to load the pre-trained MobileNetV2 ImageNet image classification model. This model was trained on the ImageNet (https://en.wikipedia.org/wiki/ImageNet (https://en.wikipedia.org/wiki/ImageNet)) benchmark image dataset that contains millions of photographic images from 1000 labeled categories. This model consists of over 150 connected "layers" (this is where the deep in deep learning comes from) that each performs computational transformations on the image data to eventually return a prediction of which label category an image belongs to. The final layer, called "predictions" in the summary output below, takes the numerical representation of the image and provides "probabilities" for each of the 1000 category classes.

In [24]:
```
mobilenet_base = tf.keras.applications.MobileNetV2(include_top=True,
                                                   weights='imagenet',
                                                   pooling='avg')
```

In [25]:

```
layer_strings = []
mobilenet_base.summary(print_fn = lambda x: layer_strings.append(x))
print('\n'.join(layer_strings[:15]))
print('\n...\n')
print('\n'.join(layer_strings[-15:]))
```

```
Model: "mobilenetv2_1.00_224"
________________________________________________________________________
__________________________
Layer (type)                    Output Shape         Param #     Connec
ted to
========================================================================
==========================
input_1 (InputLayer)            [(None, 224, 224, 3) 0

________________________________________________________________________
__________________________
Conv1_pad (ZeroPadding2D)       (None, 225, 225, 3)  0           input_
1[0][0]

________________________________________________________________________
__________________________
Conv1 (Conv2D)                  (None, 112, 112, 32) 864         Conv1_
pad[0][0]

________________________________________________________________________
__________________________
bn_Conv1 (BatchNormalization)   (None, 112, 112, 32) 128         Conv1
[0][0]

________________________________________________________________________
__________________________
Conv1_relu (ReLU)               (None, 112, 112, 32) 0           bn_Con
v1[0][0]

________________________________________________________________________
__________________________
expanded_conv_depthwise (Depthw (None, 112, 112, 32) 288         Conv1_
relu[0][0]

...


________________________________________________________________________
__________________________
Conv_1 (Conv2D)                 (None, 7, 7, 1280)   409600      block_
16_project_BN[0][0]

________________________________________________________________________
__________________________
Conv_1_bn (BatchNormalization)  (None, 7, 7, 1280)   5120        Conv_1
[0][0]

________________________________________________________________________
__________________________
out_relu (ReLU)                 (None, 7, 7, 1280)   0           Conv_1
_bn[0][0]

________________________________________________________________________
__________________________
global_average_pooling2d (Globa (None, 1280)         0           out_re
lu[0][0]

________________________________________________________________________
__________________________
predictions (Dense)             (None, 1000)         1281000     global
_average_pooling2d[0][0]
========================================================================
==========================
Total params: 3,538,984
Trainable params: 3,504,872
Non-trainable params: 34,112
```

However, for the use case of clustering, we're not necessarily after the classifications. So we remove the final layer from the model, and save this new model as "mobilenet_features".

In [26]:
```
mobilenet_features = tf.keras.Model(inputs=mobilenet_base.input,
                                    outputs=mobilenet_base.layers[-2].ou
tput)
```

Next, we create a Python function that will take an image path, resize the image to the specified MobileNetV2 input size (224 x 224 pixels), and then convert it to a 3-dimensional matrix of values (a 2-dimensional pixel value matrix for the Red, Green, and Blue color "channels" of each image).

In [27]:
```
def path_to_array(image_path):
    """
    Reads in an image from disk, resizes it using the Pillow library, co
nverts
    it to a Numpy array, and then runs MobileNetV2-specific preprocessin
g.

    Parameters
    ----------
    image_path : string
        Local file path to an image file.

    Returns
    -------
    im: Numpy array
        A processed Numpy array that is ready to be fed into the MobileN
etV2
        Keras model.
    """
    pil_image = Image.open(image_path).resize((224,224), Image.ANTIALIAS
)
    np_image = np.array(pil_image)
    np_image = np.expand_dims(np_image, axis=0)
    im = tf.keras.applications.mobilenet_v2.preprocess_input(np_image)
    return im
```

Next, we run all of our images through this model and then "stack" all of these image matrices together into a single matrix, so that we can take advantage of the model's batch prediction functionality. Finally, we feed this image matrix into the "mobilenet_features" predict function to return a 1280-value "feature vector" interpretation of the input image.

In [28]:
```
start = time.time()
images = []
for image_id in painting_ids:
    painting_path = f'saam_thumbnails/{image_id}.jpg'
    img = path_to_array(painting_path)
    images.append(img)

image_stack = np.vstack(images)
feature_vector = mobilenet_features.predict(image_stack, batch_size=32)
end = time.time()
print(end - start)
```

```
21.531198024749756
```

We can see we now have an 808 (1 entry for each of 808 filtered paintings) x 1280 (from the MobileNetV2 feature vector) matrix.

In [29]:
```
feature_vector.shape
```

Out[29]:
```
(808, 1280)
```

In [30]:
```
with open('saam_painting_feature_vectors.npy','wb') as np_out:
    np.save(np_out, feature_vector)
```

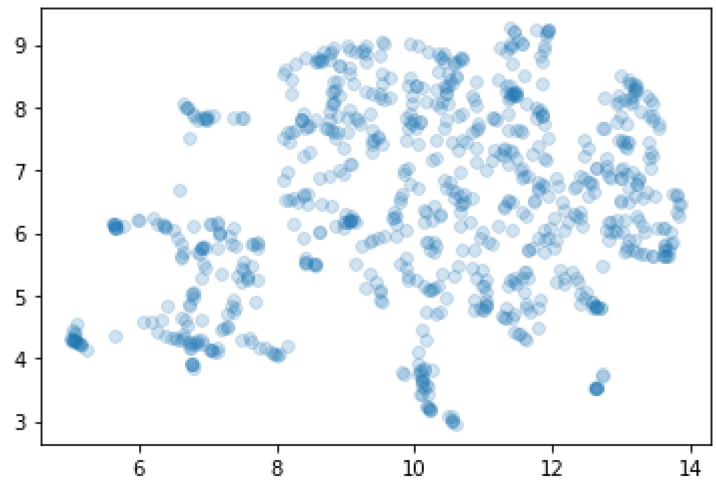
Clustering images with UMAP

Now that we have 808 feature vectors, we can process them with the UMAP algorithm to cluster similar images together. We can see in this graph, which plots each image as a semi-transparent point on a scatterplot, that there are certain areas where many paintings cluster very closely together.

In [31]:
```
embedding = UMAP(n_neighbors=6,
                 min_dist=0.01,
                 metric='correlation').fit_transform(feature_vector)

plt.scatter(embedding[:,0], embedding[:,1],
            alpha=0.2)
```

Out[31]: <matplotlib.collections.PathCollection at 0x19090e1c0>



Next, we take these image "locations" from the UMAP output, and convert these into a normalized table.

In [32]:
```
embedding_normalized = minmax_scale(embedding)
df = pd.DataFrame(embedding_normalized, index=painting_ids, columns = [
'x','y'])
df.head()
```

Out[32]:

	x	y
SAAM-1929.6.144_1	0.508578	0.707142
SAAM-1967.136.6_1	0.987404	0.580644
SAAM-1958.5.3_1	0.949186	0.685677
SAAM-1984.50_2	0.798967	0.601339
SAAM-1973.150_1	0.862443	0.093245

In [33]:
```
df = df.sample(frac=1, random_state=100)
df.head()
```

Out[33]:

	x	y
SAAM-1985.66.352_1	0.898060	0.815729
SAAM-1983.95.117_1	0.616291	0.850602
SAAM-1940.9.1_1	0.626627	0.915389
SAAM-1962.4.6_1	0.765621	0.703215
SAAM-1962.13.20A_1	0.264494	0.298959

Next, we create a 2500 x 5000 pixel blank "canvas" and then convert those UMAP locations into canvas locations, where we "paste" each of our painting images.

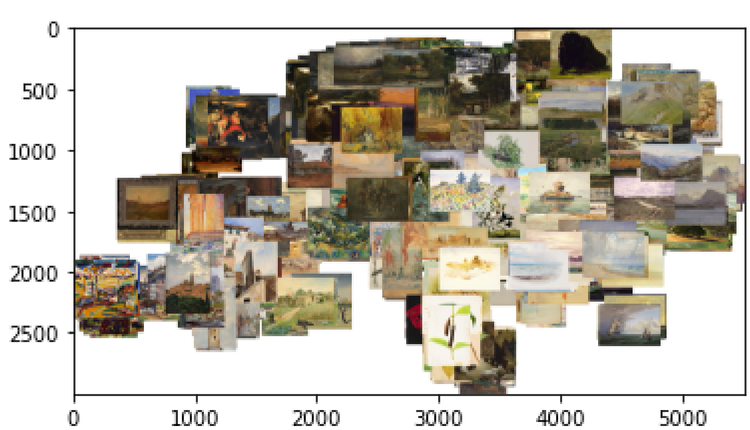
In [34]:
```
CANVAS_HEIGHT = 2500
CANVAS_WIDTH = 5000
canvas = np.full((CANVAS_HEIGHT+500,CANVAS_WIDTH+500,3),255, dtype=np.ui
nt8)

THUMBNAIL_SIZE = 500,500

for row in df[::-1].itertuples():
    image_path = os.path.join('saam_thumbnails', row.Index) + '.jpg'
    x_pos = round(row.x * CANVAS_WIDTH)
    y_pos = CANVAS_HEIGHT - round(row.y * CANVAS_HEIGHT)

    pil_image = Image.open(image_path)
    pil_image.thumbnail(THUMBNAIL_SIZE)
    np_image = np.array(pil_image)
    canvas[y_pos:np_image.shape[0]+y_pos,
           x_pos:np_image.shape[1]+x_pos] = np_image
plt.imshow(canvas)
```

Out[34]: <matplotlib.image.AxesImage at 0x190e48070>



That tiny plot in this notebook is hard to see clearly, so let's output it to file so that we can zoom in on the details … and maybe print out a poster if you like.

In [35]:
```
saam_umap = Image.fromarray(canvas)
saam_umap.save('saam_umap.png')
```

Searching for semantically similar paintings using Annoy

Now that we've "vectorized" the 808 SAAM paintings using the MobileNetV2 model, we can use these vectors as representatives to search against. This type of search can be done using a nearest neighbor algorithm. Specifically, Spotify has implemented this algorithm for quickly matching similar song vectors. They have open sourced this implementation, called Annoy (Approximate Nearest Neighbors Oh Yeah), as a Python library.

In [44]:
```
from annoy import AnnoyIndex
```

We will use the AnnoyIndex functionality to produce a feature vector search index from the vectors we produced earlier.

In [49]:
```
painting_index = AnnoyIndex(1280, metric='angular')
for idx, feature in enumerate(feature_vector):
    painting_index.add_item(idx, feature)
painting_index.build(10) #Build index with 10 trees
```

Out[49]:
```
True
```

Ok, now that we have a search index built, let's find another image to search against. For this example, I'll go back to the original flattened dataset we created (of over 12,000 objects), before we filtered to landscape paintings. Let's try to find some photos of Yosemite.

In [50]:
```
yosemite_photos = saam_df[(saam_df['object_type'] == 'Photography-Photop
rint') &
                          (saam_df['title'].str.lower().str.contains('y
osemite'))]
print(len(yosemite_photos))
yosemite_photos.head()
```

```
13
```

Out[50]:

	id	unitCode	title	media_count	media_id	t
55	edanmdm-saam_1994.91.281	SAAM	The Vernal and Nevada Falls, from Glacier Poin...	1.0	SAAM-1994.91.281_1	Landscapes|' Falls|Waterfalls|Bird's
1557	edanmdm-saam_1994.89.1	SAAM	Valley of the Yosemite from Union Point	1.0	SAAM-1994.89.1_1	Landscapes|Yos Valley|Valleys|Rivers|N
2381	edanmdm-saam_1994.91.283	SAAM	Yosemite Falls, Reflected	1.0	SAAM-1994.91.283_1	Landscapes|Waterfalls|Yos Falls|
3039	edanmdm-saam_2009.36.2	SAAM	Mirror Lake, Yosemite Valley, California	1.0	SAAM-2009.36.2_1	Landscapes|Yos Valley|Lakes|Mirro
3131	edanmdm-saam_1994.91.279	SAAM	Half Dome, Yosemite, California	1.0	SAAM-1994.91.279_1	Mountains|Landscape Dome|

There are 13 matching open access photos, but let's grab the Mirror Lake photo by Charles Roscoe Savage (EDAN ID of SAAM-2009.36.2_1). We can use a slightly modified version of the download_thumbnail function we created earlier to download this photo from AWS S3. Sure enough, that lake in the photo looks like a mirror.

In [72]:
```python
from IPython import display

edan_id = 'SAAM-2009.36.2_1'

thumb_size = (500, 500)
s3_url = f'smithsonian-open-access/media/saam/{edan_id}.jpg'
query_image = f'test_image.jpg'
with fs.open(s3_url,'rb') as s3_image:
    pil_image = Image.open(s3_image)
    pil_image.thumbnail(thumb_size)
    pil_image.save(query_image)
display.Image(query_image)
```

Out[72]:

The first thing we'll need to do is produce a 1280 MobileNetV2 feature vector for this image.

In [55]:
```python
query_array = path_to_array(query_image)
query_vector = mobilenet_features.predict(query_array).squeeze()
query_vector.shape
```

Out[55]: (1280,)

Now that we have a feature vector for this photo, we can use the "get_nns_by_vector" (nns being nearest neighbors) method from Annoy to pass in our query vector and find the 3 closest matches from the painting search index.

In [68]:

```
closest_3 = painting_index.get_nns_by_vector(query_vector, 3,
                                             include_distances=True)
closest_ids, closest_distances = closest_3
print(closest_ids)
print(closest_distances)
```

```
[192, 330, 792]
[0.7721788287162781, 0.7895978093147278, 0.7987999320030212]
```

In [69]:

```
import matplotlib.image as mpimg
```

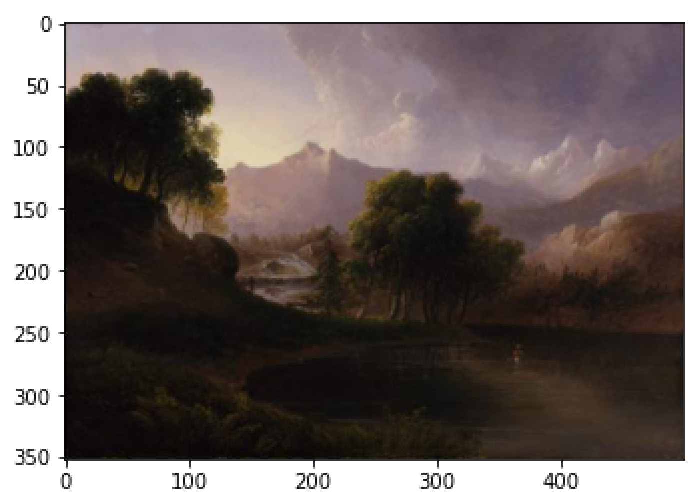
In [71]:
```
for media_order in closest_ids:
    print(painting_ids[media_order])
    match_image = f'saam_thumbnails/{painting_ids[media_order]}.jpg'
    mpl_img = mpimg.imread(match_image)

    plt.imshow(mpl_img)
    plt.show()
```

SAAM-1983.95.141_1

SAAM-1994.82_1

SAAM-1976.54_1




```
In [ ]:
```