Flood Extent Extraction

This notebook is designed to be an end to end solution that is able to extract flood extents from supplied imagery.

All configurations are contained within the `Notebook settings` cell. There are no other configurable options outside of the Notebook settings cell

NoteBook settings

All user configurable elements are contained in this cell.

- Input Image settings
  - `nrg_image_s3_source` - String: Amazon S3 source directory containing flood images destined for processing eg. s3://ss-csu-dataset/raw/Brewarrina_Flood_2021_04_15cm_NRG/
  - `nrg_image_storage_directory` - String: Storage directory where flood images will be stored for processing.
  - `image_scale` - Float: Reduce image size. This is used to boost performance. This should be set to a values approximate to the scale of images used to train the GMM model.
- GMM settings
  - `gmm_storage_directory` - String: Directory containing the pre trained GMM model
  - `gmm_flood_clusters` - Tuple: Clusters that contain flood pixels
- Contouring / polygon settings
  - `minimum_contour_size` - Int: Minimum pixel area for a contour / polygon to be considered valid
- Logging settings
  - `log_file_prefix` - String: Prefix that will be attached to all log files.
  - `log_storage_directory` - String: Location to store log files
- Output settings
  - `raster_shp_output_prefix` - String: Prefix for output shape and raster files. Output files names will also include date and time the file was created.
  - `output_directory` - String: Location to store shape files, inspection and zip file.

In [ ]:

```
#Image constants
nrg_image_s3_source = "s3://live-demo-images/brewarrina/"
nrg_image_storage_directory = "../../Images/"
image_scale = 0.1
#GMM constants
gmm_storage_directory = "../../Models/GMM/gmm_1_3_4_5_v2_master_image_nrg_0.1.gmm"
gmm_flood_clusters = (1,3,4,5)
#Contour constants
minimum_contour_size = 100 # pixels
#Log file constants
log_file_prefix = "NOTEBOOK_LOG"
log_storage_directory = "../../Logs/"
#Output settings
raster_shp_output_prefix = "demo"
output_directory = "../../Out"
```

Imports and system setup

Below cells configure the python / conda environment and performs setup tasks such as configuring the logging mechanisms. Please note it can take some time for the conda environment to completes its setup.

In [ ]:

```
#conda commands for environment setup
%conda install -c conda-forge gdal fiona rasterio

#pip commands in case conda cant get the required packages
!pip install 'opencv-python>=4.5.3.56' # required as older versions fail opening some jp2 images
```

In [ ]:

```
# imports modules - this could be moved to a seperate file once modules have stabilised.
#-~ Standard Libraries
import os
import sys
import datetime
import logging
import shutil
import traceback

#-~ Utility Libraries
import numpy as np
import matplotlib.pyplot as plt
import pickle
import boto3
try:
    from urlparse import urlparse
except ImportError:
    from urllib.parse import urlparse

#-~ Mapping
from osgeo import gdal
from osgeo import osr
import fiona
import rasterio
from rasterio.io import MemoryFile
import rasterio.merge

#-~ Computer Vision libraries
os.environ["OPENCV_IO_MAX_IMAGE_PIXELS"] = pow(2,40).__str__()
os.environ["OPENCV_IO_ENABLE_JASPER"] = "true"
import cv2
from sklearn.mixture import GaussianMixture
from skimage.transform import rescale
```

In [ ]:

```
cell_error = False
# Configure logging

# safeguard to prevent accidently running this more than once, running more than once breaks logging functionality
if not 'logger_intiated' in globals():
    logger = logging.getLogger(log_file_prefix)
    # To use differen't log level for file and console
    timestamp = datetime.datetime.utcnow().strftime('%Y%m%d_%H-%M-%S')
    formatter = logging.Formatter('[%(asctime)s] %(name)s %(levelname)s - %(message)s')
    # File settings

    filename = os.path.join(log_storage_directory , f"{log_file_prefix}_{timestamp}.log" )
    try:
        file_handler = logging.FileHandler(filename=filename)
        file_handler.setLevel(logging.DEBUG)
        file_handler.setFormatter(formatter)
        #stream stdout settings
        stream_handler = logging.StreamHandler(sys.stdout)
        stream_handler.setLevel(logging.INFO)
        stream_handler.setFormatter(formatter)
        # The handlers have to be at a root level since they are the final output
        logger.addHandler(stream_handler)
        logger.addHandler(file_handler)
        logger.setLevel(logging.DEBUG)

        # Log global debug to a seperate file. Log file will include debug logs from import modules
        # File settings
        global_filename=os.path.join(log_storage_directory , f"{log_file_prefix}_EXTRA_{timestamp}.log" )
        file_handler = logging.FileHandler(filename=global_filename)
        file_handler.setLevel(logging.DEBUG)
        file_handler.setFormatter(formatter)
        logging.basicConfig(
            level=logging.DEBUG,
            handlers=[
                file_handler
            ]
        )
        logger.info(f"Finished configuring logging. Log file: {filename}, Global log file: {global_filename}")
        logger_intiated = True
    except Exception as e:
        cell_error = True
        logger.error(f"Unable to configure logging: {e}.")
else:
    logger.warning(f"Logger is already configured. Log file: {filename}, Global log file: {global_filename}")

if cell_error:
    raise SystemExit("Execution Stopped")
```

In [ ]:

```
# Log settings to file
logger.debug(f"Notebook settings:\n"
f"nrg_image_s3_source  {nrg_image_s3_source} {type(nrg_image_s3_source)} \n"
f"nrg_image_storage_directory  {nrg_image_storage_directory} {type(nrg_image_storage_directory)} \n"
f"image_scale {image_scale} {type(image_scale)}\n"
f"gmm_storage_directory  {gmm_storage_directory} {type(gmm_storage_directory)}\n"
f"gmm_flood_clusters  {gmm_flood_clusters} {type(gmm_flood_clusters)}\n"
f"minimum_contour_size  {minimum_contour_size} {type(minimum_contour_size)}\n"
f"log_file_prefix  {log_file_prefix} {type(log_file_prefix)}\n"
f"log_storage_directory  {log_storage_directory} {type(log_storage_directory)}\n"
f"raster_shp_output_prefix {raster_shp_output_prefix} {type(raster_shp_output_prefix)}\n"
f"output_directory  {output_directory} {type(output_directory)}")
```

Download images

Download images from s3 bucket and store in directory `nrg_image_storage_directory`. Once downloaded image path will be enumerated for images of jpeg2000. Extension must be `.jp2`.

In [ ]:

```
# Downloading of images from s3 bucket will happen here
#!aws s3 cp --recursive $S3_Image_Folder ../../Images
cell_error = False

s3_client = boto3.client('s3')

def download_dir(prefix, local, bucket, client=s3_client):
    """
    params:
    - prefix: pattern to match in s3
    - local: local path to folder in which to place files
    - bucket: s3 bucket with target contents
    - client: initialized s3 client object
    """
    keys = []
    dirs = []
    next_token = ''
    base_kwargs = {
        'Bucket':bucket,
        'Prefix':prefix.lstrip('/'),
    }
    while next_token is not None:
        kwargs = base_kwargs.copy()
        if next_token != '':
            kwargs.update({'ContinuationToken': next_token})
        results = client.list_objects_v2(**kwargs)
        contents = results.get('Contents')
        for i in contents:
            k = i.get('Key')
            if k[-1] != '/':
                keys.append(k)
        next_token = results.get('NextContinuationToken')
    for k in keys:
        fn = os.path.split(k)[-1]
        dest_pathname = os.path.join(local, fn)
        logger.debug(f"Downloading {fn} to {local} from bucket {bucket}")
        client.download_file(bucket, k, dest_pathname)

# split the s3 path in to components
s3_path_components = urlparse(nrg_image_s3_source, allow_fragments=False)
# download files
logger.info(f"Started downloading all files from {nrg_image_s3_source} to {nrg_image_storage_directory}")
try:
    download_dir(s3_path_components.path, nrg_image_storage_directory, s3_path_components.netloc)
    logger.info(f"Finished downloading files from {nrg_image_s3_source}")
except Exception as e:
    logger.error(f"Unable to download images: {e}", exc_info=True)
    cell_error = True

if cell_error:
    raise SystemExit("Execution Stopped")
```

In [ ]:

```
cell_error = False

#populate list of image paths
nrg_image_paths = []

# Get image paths
try:
    # get all file names
    for file in os.listdir(nrg_image_storage_directory):
        if file.endswith(".jp2"):
            path = os.path.join(nrg_image_storage_directory, file)
            nrg_image_paths.append(path)
    # Count images
    if len(nrg_image_paths) > 0:
        logger.info(f"Directory {nrg_image_storage_directory} contains {len(nrg_image_paths)} images")
    else:
        # Raise exception if no images found
        raise Exception(f"No images in directory {nrg_image_storage_directory}")
        cell_error = False
except Exception as e:
    logger.error(f"Unable to open images: {e}", exc_info=True)
    cell_error = True

if cell_error:
    raise SystemExit("Execution Stopped")
```

Extract clusters

Below will load the 'pretrained' Gaussian Mixture Model (GMM) and extract flood clusters from the images. Clusters will then be joined to create a binary image to deterine flood extent.

In [ ]:

```
cell_error = False
# open the GMM model
try:
    with open(gmm_storage_directory, 'rb') as file:
        gmm = pickle.load(file)
    logger.info(f"Opened GMM model {gmm_storage_directory}")
except Exception as e:
    logger.error(f"Unable to open GMM model {gmm_storage_directory}: {e}", exc_info=True)
    cell_error = True

if cell_error:
    raise SystemExit("Execution Stopped")
```

In [ ]:

```
cell_error = False

binary_masks = []
hp_binary_masks = []
# loop through all images
logger.info(f"Started opening and clustering {len(nrg_image_paths)} images in {nrg_image_storage_directory}.")
for i in range(len(nrg_image_paths)):
    try:
        # build image path
        image_name = os.path.split(nrg_image_paths[i])[-1]
        # open the image
        nrg_raw_image = cv2.imread(nrg_image_paths[i])
        #nrg_raw_image = openImage(nrg_image_paths[i])
        logger.debug(f"Opened {nrg_image_paths[i]}. Image shape {nrg_raw_image.shape}")
        #scale image if required
        if not image_scale == 1.0:
            # multichannel required, maintains all channels in scaled image
            nrg_raw_image = rescale(nrg_raw_image, image_scale, anti_aliasing=False, multichannel=True)
            logger.debug(f"Scaled image {image_name} to {image_scale}. New image shape {nrg_raw_image.shape}")

        # Perform GMM clustering
        channels = nrg_raw_image.shape[-1]
        vectorized_image = nrg_raw_image.reshape((-1, channels))
        # predict clusters
        gmm_cluster = gmm.predict(vectorized_image)
        gmm_cluster = gmm_cluster.reshape(nrg_raw_image.shape[:2])
        # determine probabilities
        gmm_proba = gmm.predict_proba(vectorized_image)
        gmm_proba = gmm_proba.reshape(nrg_raw_image.shape[:2] + (gmm_proba.shape[-1],))
        logger.debug(f"Finished opening and clustering on {image_name} image")
        #joining clusters for binary image
        logger.debug(f"Joining flood clusters for {image_name}")
        joined_img = gmm_cluster == gmm_flood_clusters[0]
        hp_joined_img = np.greater(gmm_proba[:,:,gmm_flood_clusters[0]],0.15)
        for c in range(1, len(gmm_flood_clusters)):
            joined_img = np.logical_or(joined_img, gmm_cluster == gmm_flood_clusters[c])
            hp_joined_img = np.logical_or(hp_joined_img, np.greater(gmm_proba[:,:,gmm_flood_clusters[c]],0.15))
        #calculate % of image that is flood this is a rough estimation
        flood_percentage = (joined_img.sum() / joined_img.size) * 100
        hp_flood_percentage = (hp_joined_img.sum() / hp_joined_img.size) * 100
        logger.debug(f"Clustering identified {str(round(flood_percentage, 2))}% of {image_name} as flood and {str(round(hp_flood_percentage, 2))}% as high probability of flood.")
        hp_binary_masks.append(hp_joined_img)
        binary_masks.append(joined_img)

    except Exception as e:
        logger.error(f"Unable to open and process file {nrg_image_paths[i]}: {e}", exc_info=True)
        print(traceback.format_exc())
        cell_error = True

if cell_error:
    raise SystemExit("Execution Stopped")
else:
    logger.info(f"Finished opening and clustering {len(nrg_image_paths)} images")
```

Contour binary mask images

Using the binary images extracted after clustering. We will countour the edges resulting in polygons outlining captured flood areas.

In [ ]:

```
cell_error = False
# open images as geo data sets
logger.info(f"Combining images")
datasets = []
try:
    for i in range(len(nrg_image_paths)):
        logger.debug(f"Opening {nrg_image_paths[i]}")
        ds = rasterio.open(nrg_image_paths[i])
        # get dataset profile
        profile = ds.profile
        profile.update(
            dtype=rasterio.uint8,
            count=2)
        ds.close()
        # append data set to array
        logger.debug(f"Opening temporary memory file for binary image")
        memfile = MemoryFile()
        mem_ds = memfile.open(**profile)
        mem_ds.write(binary_masks[i], 1)
        mem_ds.write(hp_binary_masks[i], 2)
        datasets.append(mem_ds)
    #combine datasets and save to disk
    timestamp = datetime.datetime.utcnow().strftime('%Y%m%d_%H-%M-%S')
    raster_file = os.path.join(output_directory, f"{raster_shp_output_prefix}_{timestamp}.jp2")
    logger.debug(f"Merging {len(datasets)} binary images and writing to {raster_file}")
    rasterio.merge.merge(datasets, dst_path=raster_file, dst_kwds=profile)
    # open dataset and read flood layer
    logger.debug(f"Opening binary raster image as raster dataset")
    merged_dataset = rasterio.open(raster_file)
    logger.debug(f"Reading flood extent raster")
    flood_layer = merged_dataset.read(1)
    hp_flood_layer = merged_dataset.read(2)

except Exception as e:
    logger.error(f"Error combining images: {e}", exc_info=True)
    cell_error = True

if cell_error:
    raise SystemExit("Execution Stopped")
else:
    logger.info(f"Finished combining images")
```

In [ ]:

```
cell_error = False
try:
    contour_results = []
    hp_contour_results = []
    logger.info(f"Started contouring {len(binary_masks)} binary images")

    logger.debug(f"Contouring {nrg_image_paths[i]}")
    contours, hierachy = cv2.findContours(np.uint8(flood_layer), cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
    # eliminate contours less than minimum area
    for c in range(len(contours)):
        if cv2.contourArea(contours[c]) > minimum_contour_size:
            contour_results.append(contours[c])
            # reverse orientation to create holes in shape
            if hierachy[0][c][3] != -1:
                contour_results[-1] = np.flipud(contour_results[-1])

    hp_contours, hp_hierachy = cv2.findContours(np.uint8(hp_flood_layer), cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
        # eliminate contours less than minimum area
    for c in range(len(hp_contours)):
        if cv2.contourArea(hp_contours[c]) > minimum_contour_size:
            hp_contour_results.append(hp_contours[c])
            # reverse orientation to create holes in shape
            if hp_hierachy[0][c][3] != -1:
                hp_contour_results[-1] = np.flipud(hp_contour_results[-1])

    logger.info(f"{len(contour_results)} contours, {len(contours) - len(contour_results)} eliminated")
    logger.info(f"{len(hp_contour_results)} high probability contours, {len(hp_contours) - len(hp_contour_results)} eliminated")

except Exception as e:
    logger.error(f"Error contouring images: {e}", exc_info=True)
    cell_error = True

if cell_error:
    raise SystemExit("Execution Stopped")
else:
    logger.info(f"Finished contouring / creating polygons")
```

Shape file

Create ESRI shape files using polygons created from the contouring process.

In [ ]:

```
def pixel2location(dx,dy):
    '''Convert pixel coordinates to spatial coordinates
    dx: x axis pixel
    dy: y axis pixel
    '''
    x = dx * pixel_x_size + pixel_x_offset
    y = dy * pixel_y_size + pixel_y_offset
    return x,y

# define shp schema
schema = {
    'geometry':'MultiPolygon',
    'properties':[('tag','str')]
}

cell_error = False

logger.info("Started converting contours / polygons to shape file")

# get the transform data
transform = merged_dataset.transform
pixel_x_offset = transform[2]
pixel_x_size = transform[0]
pixel_y_offset = transform[5]
pixel_y_size = transform[4]
logger.debug(f"Spatial data: X pixel width = {pixel_x_size}m, Y pixel height = {pixel_y_size}m, pixel (0, 0) location = {pixel_x_offset}, {pixel_y_offset}")

# loop through each contour
if len(contour_results) > 0:

    logger.debug(f"{len(contour_results)} polygons will be created")
    # Convert pixel data points to spatial
    multi_polygon = []
    for contour in range(len(contour_results)):
        polygon = []
        multi_polygon.append(polygon)
        # convert polygon pixel points to spatial points and store
        for pixel in contour_results[contour]:
            #convert
            x, y = pixel2location(pixel[0][0], pixel[0][1])
            #store
            polygon.append([x, y])

    hp_multi_polygon = []
    for contour in range(len(hp_contour_results)):
        polygon = []
        hp_multi_polygon.append(polygon)
        # convert polygon pixel points to spatial points and store
        for pixel in hp_contour_results[contour]:
            #convert
            x, y = pixel2location(pixel[0][0], pixel[0][1])
            #store
            polygon.append([x, y])

        # write to shape file
    shp_file_output = os.path.join(output_directory, f"{raster_shp_output_prefix}_{timestamp}.shp")
        #write shape file
    logger.debug(f"Writing {shp_file_output}")
    try:
        with fiona.open(shp_file_output, 'w', 'ESRI Shapefile', schema=schema, crs=merged_dataset.crs) as shp_file:
            logger.debug(f"Shape file espg: {shp_file.crs}")
            shp_file.write({
                'geometry' : {'type':'MultiPolygon',
                'coordinates': [multi_polygon]}, # Here the xyList is in brackets
                'properties': {
                    'tag': 'standard probability flood extent'
                }
            })

            shp_file.write({
                'geometry' : {'type':'MultiPolygon',
                'coordinates': [hp_multi_polygon]}, # Here the xyList is in brackets
                'properties': {
                    'tag': 'high probability flood extent'
                }
            })
    except Exception as e:
        logger.error(f"Unable to write shape file {shp_file_output}: {e}", exc_info=True)
        cell_error = True
else:
    logger.debug(f"No polygons for {nrg_image_paths[i]}. No shape file will be generated")

if cell_error:
    logger.warn(f"Finished converting polygons to shape file with errors")
else:
    logger.info(f"Finished converting polygons to shape file")
```

Compress and clean up

Compress (zip) the output files and remove any files created or downloaded onto the local machine exc luding zip files and logs.

In [ ]:

```
try:
    x = datetime.datetime.now()
    zip_file_name = (f"{x.year}{x.month}{x.day}{x.hour}{x.minute}_Outputs")
    zip_file = os.path.join(f"{output_directory}/../", zip_file_name)
    logger.info(f"Starting compression of {output_directory}, writing to {zip_file}.zip")
    shutil.make_archive(f"{zip_file}", 'zip', output_directory)
    logger.info(f"Success compressing and writing outputs to {zip_file}.zip")
    logger.info(f"Cleaning up")
    logger.info(f"Removing files from images storage {nrg_image_storage_directory}")
    for file in os.listdir(nrg_image_storage_directory):
        if file.endswith(".zip") or os.path.isdir(file):
            pass
        else:
            os.remove(os.path.join(nrg_image_storage_directory, file))
    logger.info(f"Removing files from outputs storage {output_directory}")
    for file in os.listdir(output_directory):
        if file.endswith(".zip") or os.path.isdir(file):
            pass
        else:
            os.remove(os.path.join(output_directory, file))
except Exception as e:
    logger.error(f"Unable to compress and clean up: {e}", exc_info=True)
```