G5

PROJECT DOCUMENTATION

Research & Development

Aerial Imagery Initiative

© 2021 ITC309 Group 5 CSU - Patrick, Darren, Cameron, Adam, Andrew

Confluence

1. Image Processing Methods . . . 2
1.1 Gaussian Mixture Models . . . 3
1.2 NDWI . . . 7
1.3 Mask for Clusters . . . 11
1.4 JP2 to PNG conversion from AWS s3 bucket . . . 16
1.5 Scoring and accuracy . . . 18
1.6 Clustering Comparisons . . . 19
1.7 Image Pre Processing . . . 21
1.8 U-Net (Semantic Segmentation) . . . 24
1.9 Combining Images / ShapeFiles . . . 33
2. Image annotation . . . 35
3. Knowledgebase . . . 36
3.1 Configuring AWS CLI/Jupyter for Windows . . . 37
3.2 Image resizing . . . 41
4. Supervised Semantic Segmentation . . . 42
4.1 Introduction to Supervised Machine Learning . . . 43
4.2 Data Preparation . . . 45
4.3 Artificial Inflation of Training Data-set through image augmentation . . . 50
4.4 AWS SageMaker Semantic Segmentation Results . . . 53
4.5 XGBoost & Hybrid Approaches . . . 68

Image Processing Methods

These pages document the various image processing methods that have been tested by the CSU Student Team for the Aerial Imagery Initiative. The approaches used by the team include unsupervised clustering algorithms, remote-sensing derived indices, and machine learning models. Unsupervised clustering algorithms iteratively sort a data structure into a pre-determined number of groups based on the similarity of their values. In the case of K-Means the similarity is the Euclidean distance between two or more data points and the final output are known as ‘hard assignments’. In a Gaussian Mixture Model, rather than definitively assigning each data point to a cluster, a generative probability distribution is calculated that gives a weight, or probability, that each point was generated by each Gaussian (i.e. Normal) distribution. These are referred to as ‘soft assignments’. The Normalized Difference Water Index (NDWI) is a quotient calculated from the Green and Near-Infrared channels to delineate open water features in aerial imagery. <Machine Learning>. A more detailed discussion of each method is available in their respective sections.

Testing the accuracy of each method on the entirety of the Hawkesbury-Nepean and Brewarrina flood datasets is prohibitively time-consuming. Instead, a subset of images have been selected that include various topographical features that are representative of the challenges for generalised flood-extent mapping such as cloud cover and sparse areas of vegetation protruding from flood waters. A truth image was manually generated for these test images, allowing for a simple calculation of each method’s accuracy.

Gaussian Mixture Models

A Gaussian Mixture Model is an unsupervised clustering algorithm. It is creates a generative probability distribution of multiple Gaussian clusters (i.e. normal distributions). The number of Gaussian distributions is analogous to the number of clusters defined in the K-Means clustering algorithm. Each Gaussian is described by its mean, variance, and weight. The probability density function below describes the probability of observing some input *x* given some specific Gaussian distribution. A semantic point of difference to note between K-Means and Gaussian Mixture Models is that assignments of pixels to clusters are based on probabilities, each pixel has a probability between 0 and 1 inclusive of being generated by each Gaussian distribution. For each pixel, the sum of probabilities associated with each Gaussian is 1. Pixels are assigned to the Gaussian distribution with the highest probability of generating them. For our purposes we can treat it as if it were a hard assignment as in K-Means clustering. In the context of ouralgorithm, it describes the probability that some input belongs to a specific Gaussian distribution, i.e. the probability that some pixel belongs to this distribution.

$$\mathcal{N}(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

This project uses sklearns GMM python package: https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html

Tests and results

Colour channel combinations

branch / notebook / commit

adam1 / 2.4.2_CV_GMM_colour_spaces.ipynb / 822f2f6

Results:

Test Description	Explore the effectivness of Gaussian MIxture Model clustring on different colour spaces and mixtures of colour spaces		
Branch / Notebook	adam 1 / 2.4.2 _CV_GMM_colour_spaces.ipynb		
Results commit	822f2f6		
Notes	sklearn GMM, n_components = 11, random_state=5, images scaled to 0.1, using master_image_v1		
	Channels tested	Flood clusters	Jaccard score (high is best)
	(N)IR	10 , 9 , 8 , 7 , 5, 4 , 1	0.176
	NRG	0, 1, 5	0.691
	HSV derived from NRG	0, 3	0.527
	H derived from NRG	0, 9	0.293
	HS derived from NRG	0	0.61
	RGB	0 1 4 7 10	0.224
	HSV derived from RGB	0 1 3 4	0.19
	NRGB	0 3 10	0.692
	NHSV derived from RGB	0 1 4	0.259
	NHS derived from RGB	0 3 7	0.154

Both NRGB and NRG performed well and have comparable results.

The creation of the NRGB data sets require access to all 4 bands. DCS has provided images of both NRG and RGB meaning creation of NRGB dataset requires extraction of the B channel from RGB. Supplied RGB images are in a ECW format and require conversion before use. This adds more time to the processing to create a NRGB image on top of additional clustering time of 4 channel images.

Because of this NRG is the preferable option.

Implementation Method/s

Basic implementation of skilearn GMM

1. GMM fitting/training to chosen representative image, representative image should provide enough data to represent all possibilities in the image set (cloud shadow, dark areas, etc.)

```
gmm = GaussianMixture(n_components=n)
gmm.fit(vectorized_image)
```

2. Predicting clusters on same representative image.

```
gmm_cluster = gmm.predict(vectorized_image)
```

3. Manually identify flood cluster indices, if flood clusters contain other elements such as streets or buildings, n_components can be increased until either the flood is separated from other elements or skilearn GMM advises that it is unable to find enough data to fill n_components.

4. Join flood clusters to create a single combined flood cluster.

```
joined_cluster = np.logical_or(gmm_cluster == a, gmm_cluster == b)
```

5. Reuse the trained gmm object to predict other images in the set.

```
gmm_cluster = gmm.predict(other_vectorized_image)
```

6. Repeat step 4 using cluster indices identified in step 3.

The basic implementation works however it has a few problems:

1. Trained/fitted GMM is specific to the image data set used
2. Cluster indices specific to the trained GMM (changes with each iteration)
3. Fitting GMM can be time-consuming
4. GMM not reusable

Advanced implementation skilearn GMM

Building on the techniques in Basic implementation of skilearn GMM and to address the shortcoming involved. The following improvements have been developed:

1. Fitting of the GMM is performed on a composite image. The composite image is a combination of images covering multiple datasets and variations in the the datasets
2. Flood cluster indices are identified and recorded.
3. GMM object is serialised and written/saved to disk using pickle
4. The serialised GMM model can be read from disk and used in any GMM based flood classification pipeline as the cluster indexes are already identified.

This has been implemented in the following notebook

https://bitbucket.org/csu-spatialservices/ss-datascientist-platform/src/adam1/notebooks/complete/0.1_Flood_Extent_Extraction.ipynb?viewer=nbviewer

Accuracy / Resolution

For the improvement of processing speed and having the ability to run GMM on realistic hardware configurations, images will be scale down from original size. Below is a comparison of such scaling and the effective output dIfference. It is deemed that higher resolution images give better image detail and define flood extent more accurately however comes at a greater computational and time complexity.

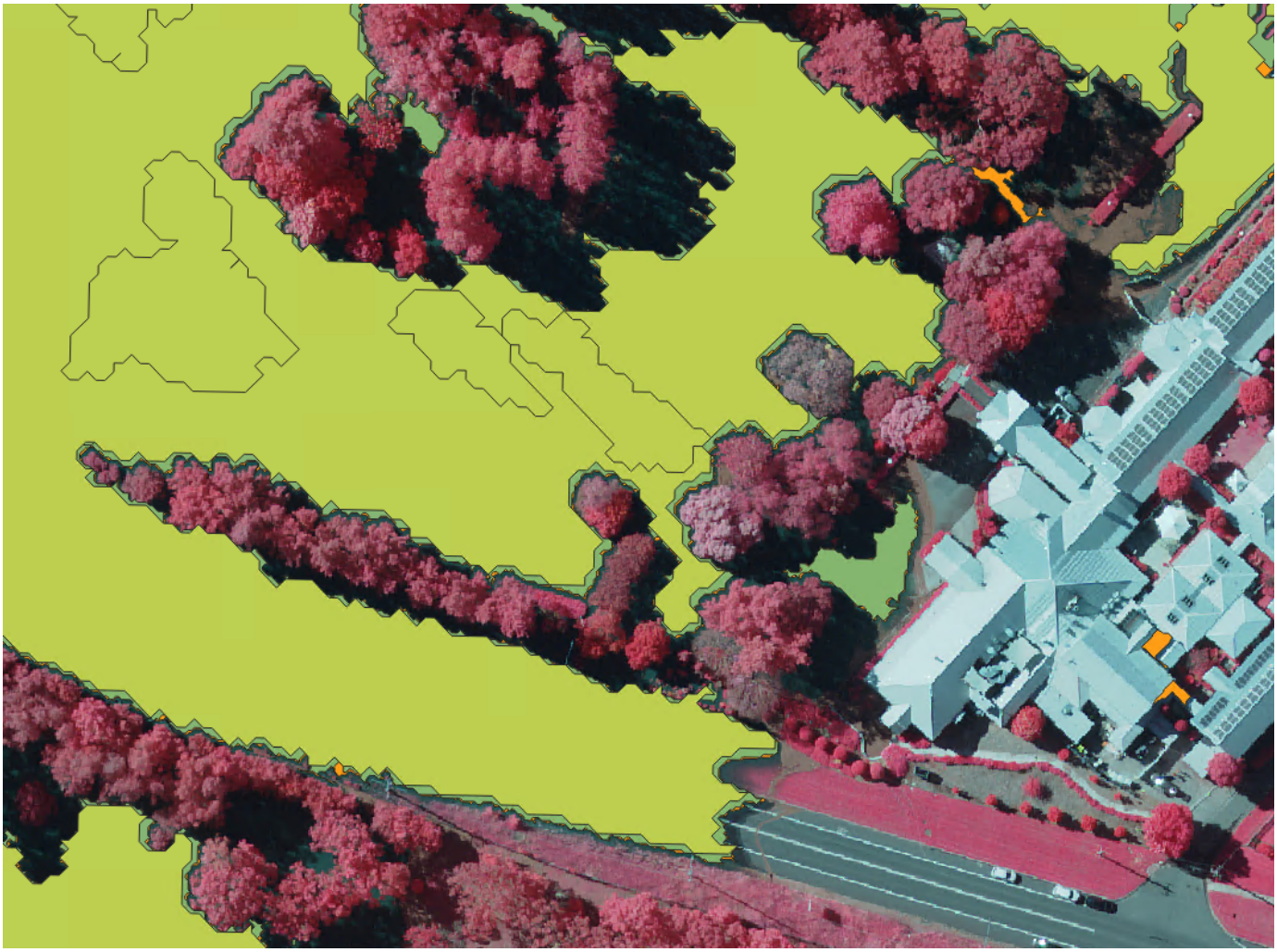
The experiments were conducted using the Hawkesbury set of 144 images of 10000px * 10000px. Times have been recored using a PC with 6 core CPU (Ryzen 5 3600), 16GB RAM

The 3 scales tested are:

1. 0.1 (10% of original) giving a resolution of 1.5m per pixel. Execution time: 25mins 41s
2. 0.3 (30% of original) giving a resolution of 0.5m per pixel. Execution time: 52mins 16s
3. 0.5 (50% of original) giving a resolution of 0.3m per pixel. Execution time: 1hr 19mins 51s

The following image displays the resulting shape file from processing images base on their respective resolution. The 0.1 data is placed on top, the 0.3 data layered in the middle and 0.5 as the bottom (barely visible). This allows comparison of the effective gain of a using higher resolutions.

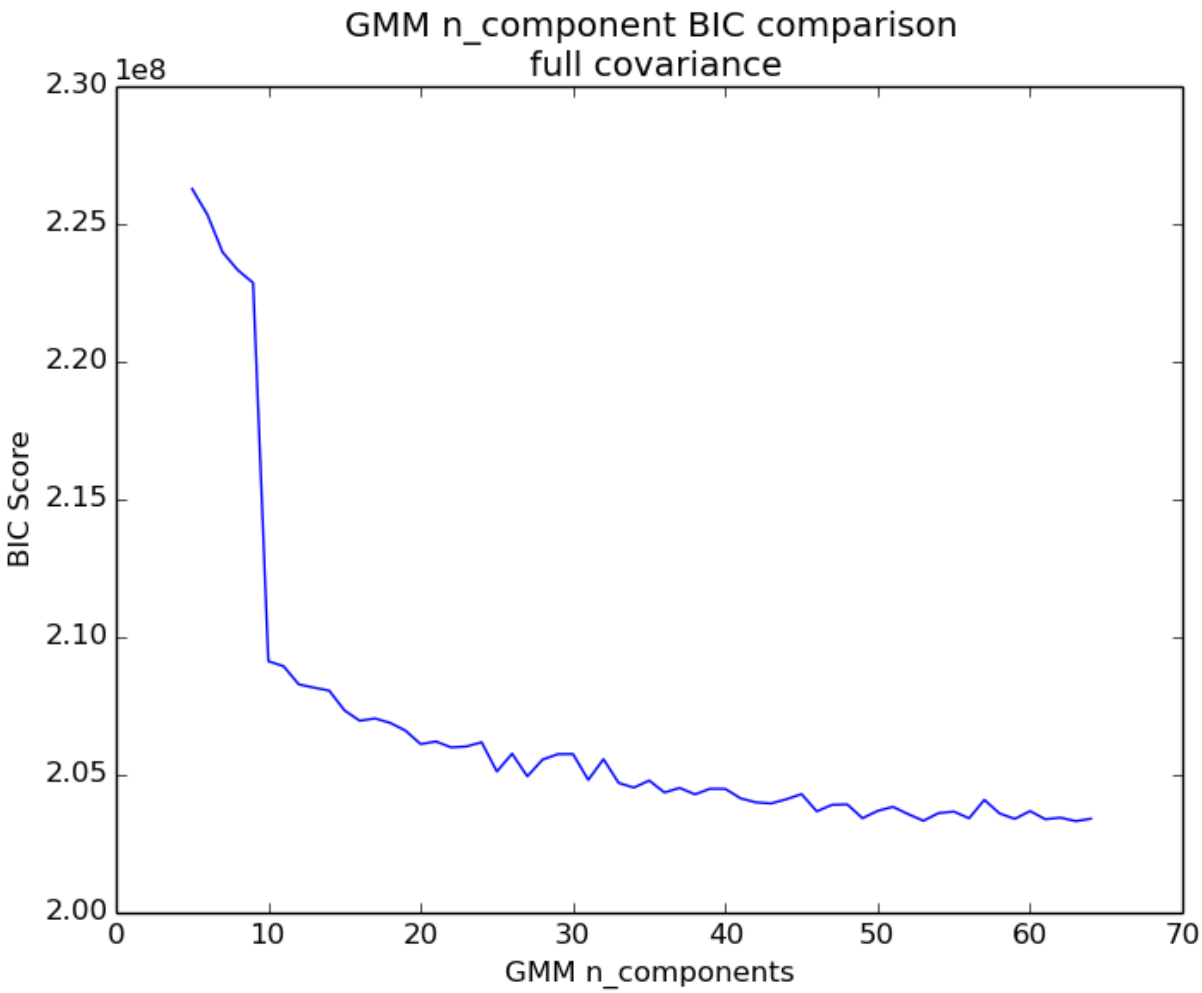
There is a noticeable difference between 0.1 to 0.3. Where 0.3 captures approximately 1 - 1.5m extra of the flood extent. 0.5 scaling offers little to no benefit. Its does capture some of the missed flood extent however this is minor.



n_component selection

Selection of n_component value has an impact on the overall performance of the GMM. To low of a value can result in a mix of image features grouped to gather (eg. flood + road). To high of a value can result in needed processing time.

Below is a graph of **Bayesian information criterion** scores of a range of n_component values (5-64). Tests were conducted with full covariance as early tests indicated it reliably resulted in lower scores.



Using below as a guide, the optimal n_component value will be approximately 20.

https://towardsdatascience.com/gaussian-mixture-model-clusterization-how-to-select-the-number-of-components-clusters-553bef45f6e4

Large single image processing

In the pursuit to perform clustering on one complete mosaic the system consumed approximately 500GB of ram for the Hawkesbury NRG image (179590px, 72510px) and run time was approximately 1hr. This test was completed on a ml.r5.24xlarge instance.

Tests were attempted on smaller instances however the Jupiter process would be killed by the OS without any python exception. This was confirmed by attempting to open the image with CV2 using plain python. The process would eventually fail with a single “killed” message.

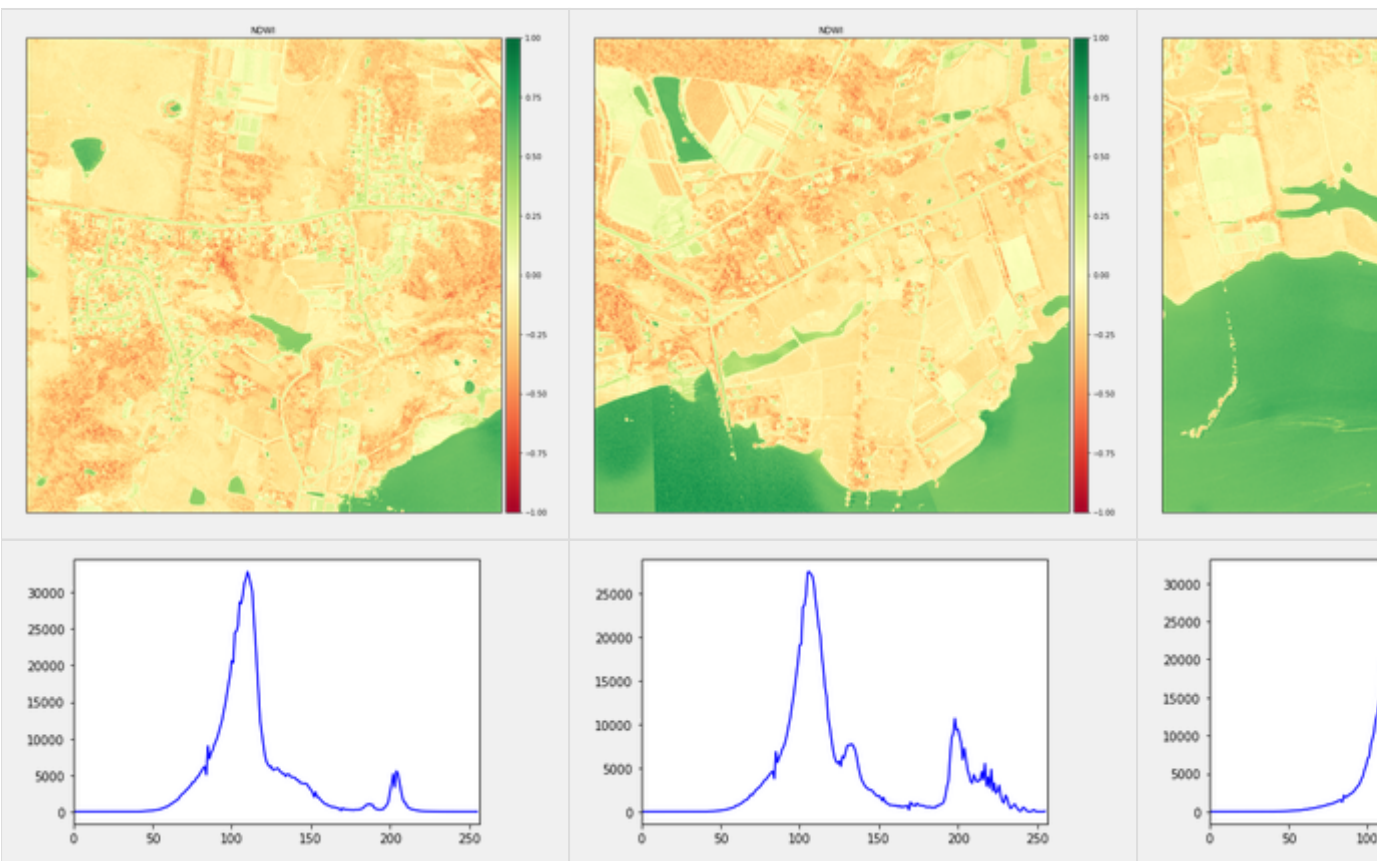
NDWI

The Normalized Water Difference Index (NDWI) was developed by McFeeters (1996) to isolate open water features from aerial images. It calculates the quotient of the difference of the green channel and the near-infrared (NIR) channel by the sum of the green channel and the near-infrared channel. Values greater than 0 are associated with water, while values near 0 or less than 0 are associated with land, soil, and vegetation (McFeeters, 1996; McFeeters, 2013). This is due to the low NIR reflectance values of water and the higher NIR reflectance values of terrestrial vegetation. The total range of values is between -1 and 1 inclusive.
The NDWI formula is:

```
(green - near infrared) / (green + near infrared)
```

One of the challenges in using NDWI is determining a cut-off value to separate water features from non-water features. McFeeters (1996) set the threshold value at 0, however initial examination of the NDWI output for this project demonstrated this is an unsuitable cut-off value. A time-series study of glacial lakes in Pakistan created land-water thresholds through trial and error as well as observations of histograms. The study determined that lake surface NDWI values ranged from 0.46 to 0.75 (Amin et al, 2019). Another approach is to determine threshold values using an unsupervised algorithm such as Otus's method, an optimisation algorithm that iteratively searches for the maximum variance between two groups for multiple threshold values. The two groups represent the target object and the background. This method is ideal for images with a bimodal intensity histogram with a well-defined valley, providing an ideal candidate threshold value to separate an object from its surroundings (Otsu, 1979). It should be noted however that the NDWI is not just a measurement strictly for the delineation of open water features. It is also used for estimating soil moisture and canopy water content (Serrano et al, 2019). This may complicate attempts to delineate flood extents when they are nearby vegetation with high moisture content. The modified Normalized Difference Water Index (mNDWI), which uses the short-wave infrared (SWIR) reflectance instead of near-infrared reflectance, provides more stable thresholds (k = 0) for comparing different images since soil and vegetation have similar reflectance in the green and SWIR bands (Mohammadi et al, 2017). Repeating the process with SWIR instead of NIR reflectance values may provide better results.

NDWI Output & Histogram of transformed values



Below are the Intersection Over Union (IOU) scores for 6 images from the Hawkesbury-Nepean and Brewarrina datasets with multiple threshold values. After running each image through the NDWI method with each threshold value, the result was then passed into the OpenCV contouring methods. A reduced set of these contours were then selected using the reduce_contours() method from 2.3_CV_Adam_Contours.ipynb. These contours were then transformed into a binary image for comparison against truth images. The IOU is used to measure the accuracy of the output. It is a ratio of the overlap between the output and the truth image over the union of the output and the truth image. To calculate this, the

intersection is calculated as the logical conjunction of the output image and the truth image and the union is calculated as the logical disjunction of the output image and the truth image. A simple comparison of each corresponding pixel value yields the desired ratio. An implementation of this can be found in 4.1_CV_Andrew_NDWI.ipynb.

One advantage of the NDWI over the GMM approach is that the presence of cloud cover has little effect on the results and is computationally more efficient than predicting clusters from a trained model and stitching together flood clusters. The drawback with this method is finding an appropriate threshold value. Although the manually selected value of k=0.3 has a reasonable performance for the sampled images, there is no guarantee it will provide the same level of performance for the rest of the dataset or for future datasets.

From a purely observational perspective. NDWI does capture the flood while reducing other features. However thresholding is required to segment to flood area from non flood and the threshold values are custom depending on the flood image set.

Further work could be down to automatically determine threshold values based on image colour ranges. There is an attempt at this in 2.5.1 _CV_Adam_NDWI .ipynb / 822f2f6.

The earthy package provides functionality to calculate NDWI.

https://earthpy.readthedocs.io/en/latest/gallery_vignettes/plot_calculate_classify_ndvi.html

Examples:

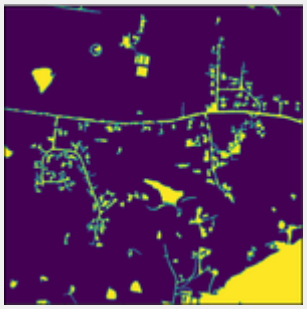
branch / notebook / commit

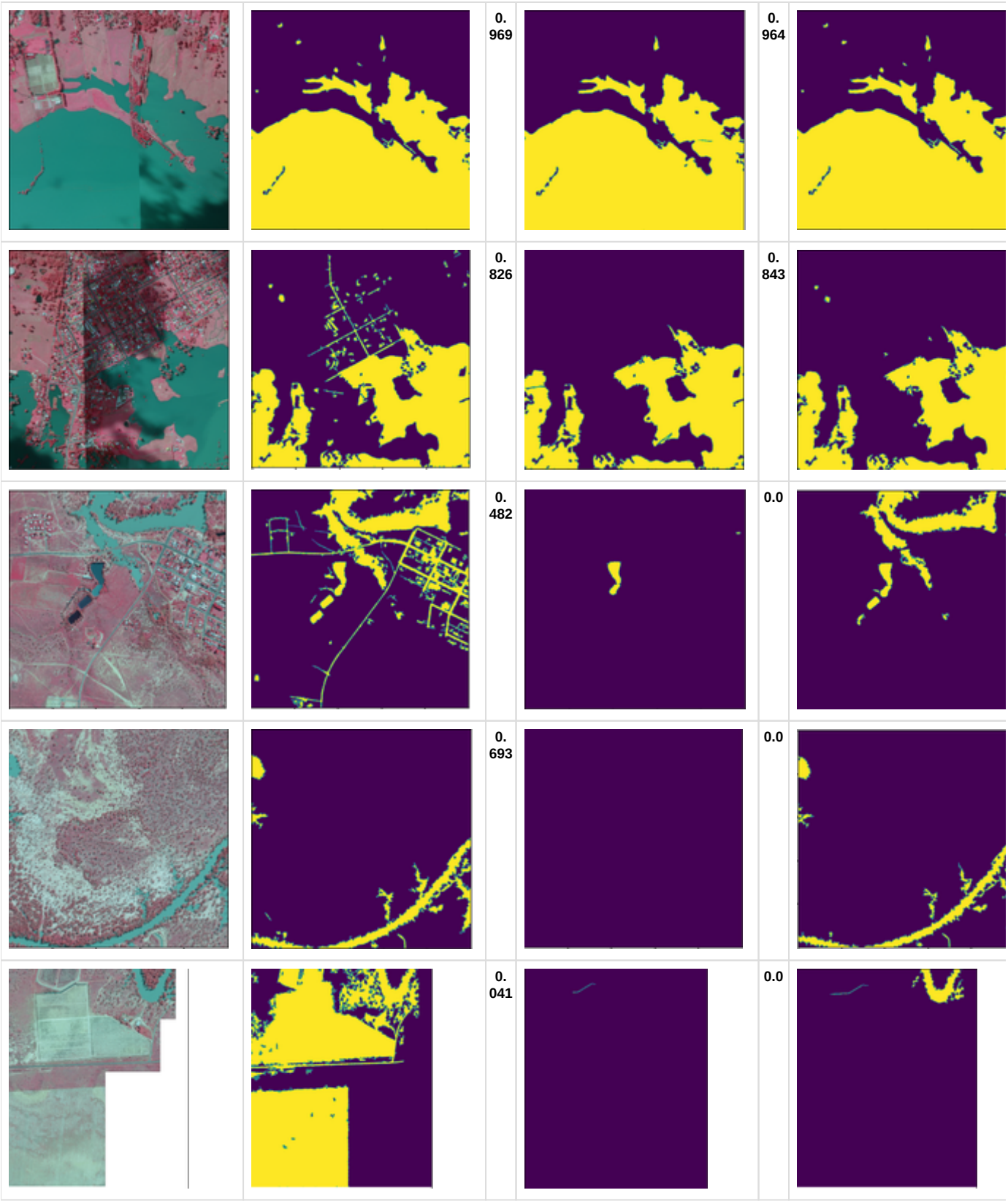
adam1 / 2.5.1_CV_Adam_NDWI .ipynb / 822f2f6

adam1 / 2.5.2 Earthpy NDWI.ipynb / 719dd0a

andrew_test / 4.0_CV_Andrew_NDWI.ipynb /

andrew_test / 4.1_CV_Andrew_NDWI.ipynb / ------

Input Image	***Otsu's Method***	***IOU***	***Time Series Study (k=0.45)***	***IOU***	***Manually Selected (k=0.30)***
		**0.335**		**0.785**	
		0.855		**0.909**	



References

Amin, A., Bano, D., Hassan, S., Goheer, M. A., Khan, A. A., Khan, M. R. & Hina, S. M. (2019). Mapping and monitoring of glacier lake outburst floods using geospatial modelling approach for Darkut valley, Pakistan. *Meteorological Applications,* 27(1), 1-12

McFeeters, S. (1996). The use of the Normalized Water Difference Index (NDWI) in the delineation of open water features. *International Journal of Remote Sensing*, 17(7), 1425-1432

McFeeters, S. (2013). Using the Normalized Difference Water Index (NDWI) within a Geographic Information System to Detect Swimming Pools for Mosquito Abatement: A Practical Approach. *Remote Sensing*, 5, 3544-3561

Mohammadi, A., Costelloe, J., F. & Ryu, D. (2017). Application of time series of remotely sensed normalized difference water, vegetation and moisture indices in characterizing flood dynamics of large-scale arid zone floodplains. *Remote Sensing of Environment*, 190, 70-82

Otsu, N. (1979). A Threshold Selection Method from Gray-Level Histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1), 62-66

Serrano, J., Shahidian, S. & da Silva, J. M. (2019). Evaluation of Normalized Difference Water Index as a Tool for Monitoring Pasture Seasonal and Inter-Annual Variability in a Mediterranean Agro-Silvo-Pastoral System. *Water*, 11(1), 62-80

Mask for Clusters

NOTE this method was abandoned due since the GMM model consistently grouped flood clusters into the same indices 1, 3, 4, 5.

As part of the process to automate the selection of flood clusters generated by the Gaussian Mixture Model and K-Means clustering algorithm, each cluster is masked back onto an image. The aim is to have each flood cluster projected back onto a body of water which, in theory, should be easy to detect algorithmically by looking at the range of its values in either RGB or HSV color spaces. However, the presence of clouds in some images complicates this step.

Figure 1 - flood area with cloud cover

```
for each cluster:
    count proportion of pixels with blue as the strongest channel
    if proportion of pixels with blue as strongest channel exceeds threshold:
        IS flood cluster
    else:
        NOT flood cluster
```

Figure 2 -pseudo-code for selecting flood cluster ** SHOULD BE GREEN CHANNEL NOT BLUE CHANNEL

Clearly, using *figure 1* as input would not produce the desired result as the presence of cloud cover in the bottom left corner differentiates a subset of the flood from the rest of the flooded area. It is therefore necessary to produce a false color space so that the flooded region appears as a homogenous area. Converting RGB images to the HSV color space and minimising the Value channel for each pixel did not produce useful results.

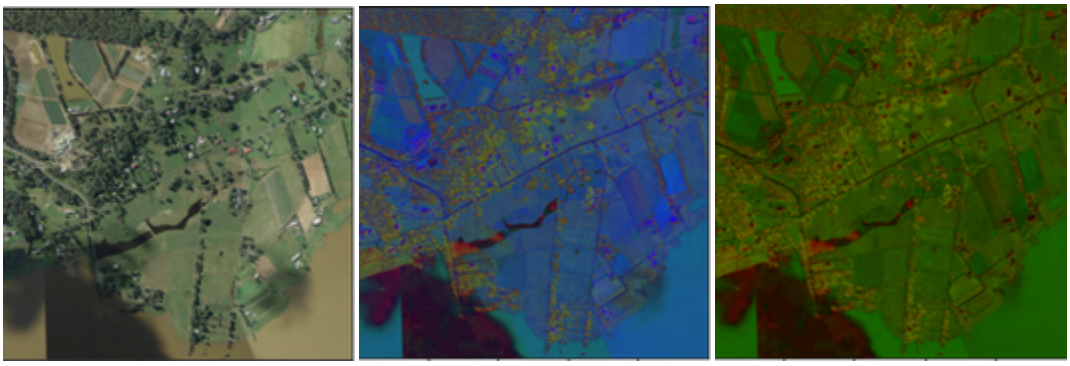


Figure 3 - left: RGB, centre: RGB converted to HSV, right: HSV with V=0 for each pixel

A candidate mask can be obtained by converting the NRG image to HSV, reducing the Value channel to 0, and then converting the resulting image to BGR.

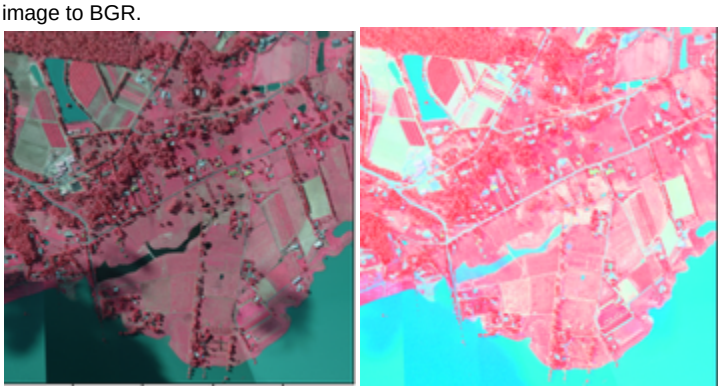


Figure 4 - left: NRG, right: NRG to HSV (V=0) to BGR

Below are the results for Gaussian Mixture Models with 4 and 11 components respectively with arbitrary threshold value 0.65.

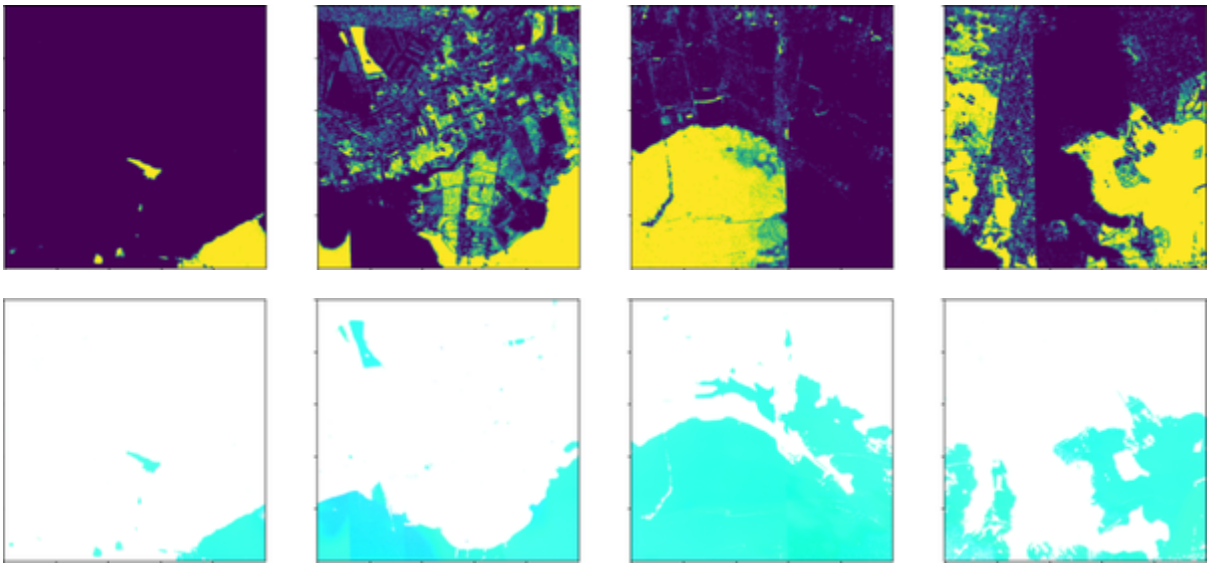


Figure 5 - automatic flood cluster selection; top: n_components=4, bottom: n_components=11

Interestingly, the algorithm produced more accurate results for a lower number of Gaussian distributions, although there was a degree of noise present in the output images. Part of the reason for the loss of accuracy for higher numbers of distributions is due to the effect of cloud cover on water regions. Even when converting the images to the HSV color space and normalising the value channel for each pixel, the flood area does

not contain homogenous pixel values. This is evident when passing the mask images through a filter that maximises the R, G, or B channels of each pixel, depending on which channel has the highest value.

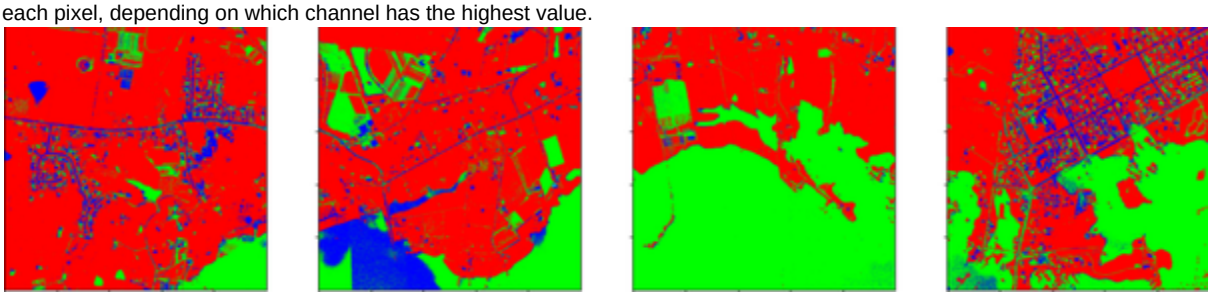
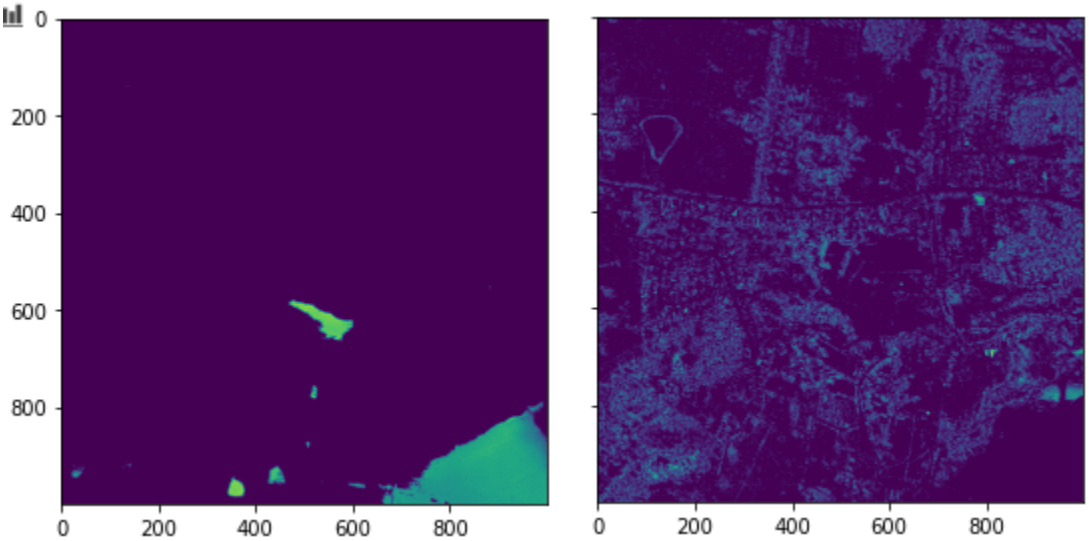


Figure 6 - max RGB filter applied to mask images

As can be seen in the bottom left corner of the 2nd and 4th images in figure 6, the presence of cloud cover creates significant variation in the flood pixel values, even after conversion and manipulation in the HSV color space.
Given the inaccurate results produced from compounded color space masks, a simpler mask was tested. According to the equation of the Normalised Difference Water Index (NDWI) water tends to have low reflectance values in the near-infrared channel. Thus, a simple mask with one channel for near-infrared values was selected. An approach similar to the pseudo-code in figure 2 was implemented. First, the output of the GMM was projected back onto the mask images. Then, the proportion of pixels with NIR values of less than 50 was calculated for each mask. The threshold value of 50 was selected from examining a histogram plot of pixel values for each mask and comparing known flood clusters against known non-flood clusters.




```
cluster: 1.1
```

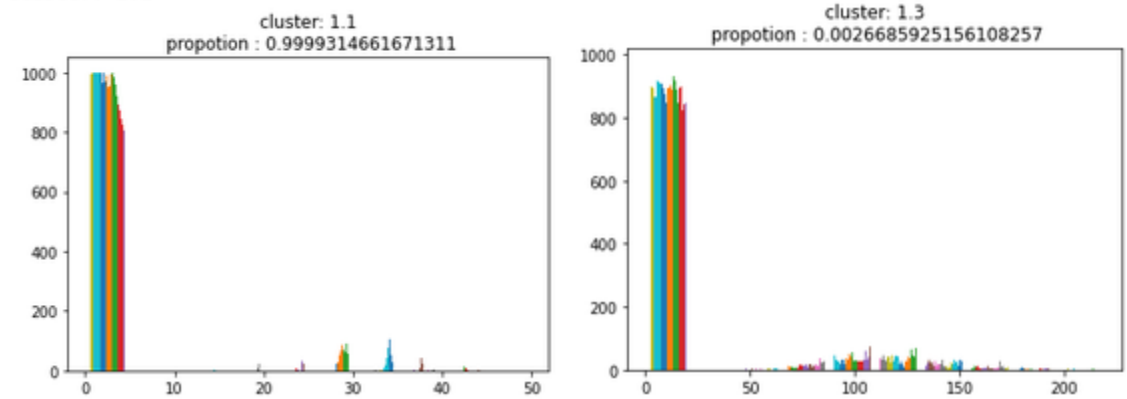


Figure 7 - top left: NIR mask of flood cluster; top right: NIR mask of non-flood cluster; bottom left: histogram of pixel values for flood cluster; bottom right: histogram of pixel values for non-flood cluster

Next, the second threshold of what proportion of pixels with NIR values less than 50 for a mask indicates a flood needed to be chosen. Table 1 lists the proportion of such pixels for each mask. Known flood clusters are highlighted.

Image/Cluster	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
1	0.999	0.000	0.003	0.411	0.000	0.000	0.272	0.000	0.285	0.355	0.000
2	0.991	0.000	0.000	1.000	0.000	0.000	0.294	0.000	0.258	0.667	0.000
3	1.000	0.000	0.090	1.000	0.047	0.000	0.998	0.000	0.163	0.000	1.000
4	0.999	0.000	0.812	0.098	0.007	0.153	0.009	0.482	0.000	0.129	1.000

Table 1 - Proportion of pixels with NIR values < 50 (rounded to 3 dp)

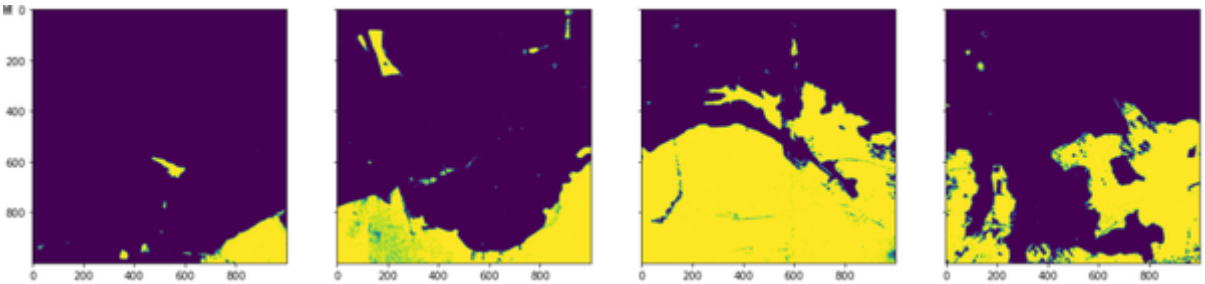


Figure 8 - Stitched flood clusters using NIR mask

Rather than having some threshold for the proportion of green-channel dominant pixels for a given cluster, the selection algorithm could use some range of green values as the criterion. However, experimentation for this is time-consuming. Instead, the Normalised Difference Water Index (NDWI) was considered as a more efficient alternative.

References

McFeeters, S. K. (1996). The use of the Normalized Water Difference Index (NDWI) in the delineation of open water features. *International Journal of Remote Sensing*, *17*(7), 1425-1432

JP2 to PNG conversion from AWS s3 bucket

Some locations need to be manually set.
NOTE: Needs to be used in an authenticated notebook environment.

```
#-- IMPORTS
import boto3
from PIL import Image
import os, re, os.path

#-- Setup s3 bucket connection
s3_client = boto3.client("s3")
s3_resource = boto3.resource("s3")

#-- Bucket settings
s3_bucket = "ss-csu-dataset"
site = "Brewarrina" #"HawkesburyCentral"
s3_key_path = f"tiles/{site}/CIR/"

#-- Download from s3
path = "image_conversion/ss/images"

def get_s3_keys(bucket):
    """Get a list of keys in an S3 bucket."""
    keys = []
    resp = s3_client.list_objects_v2(Bucket=bucket)
    for obj in resp['Contents']:
        if s3_key_path in obj['Key']:
            keys.append(obj['Key'])
    return keys

!mkdir -p image_conversion/ss/images

images = get_s3_keys(s3_bucket)
images_to_convert = []
for key in images:
    s3_image_name = key.replace(s3_key_path, "")
    images_to_convert.append(s3_image_name)
    print(s3_image_name)
    s3_client.download_file(s3_bucket, key, f"{path}/{s3_image_name}")

#-- Image conversion
converted = "image_conversion/ss/converted"
!mkdir -p image_conversion/ss/converted

if images_to_convert != None:
    for i in images_to_convert:
        img_path = (f"{path}/{i}")
        img = Image.open(img_path)
        remove_ext = i.replace(".jp2", "")
```

```
        img.save(f"{converted}/{remove_ext}.png")
        s3_client.upload_file(f"{converted}/{remove_ext}.png",
'team5tempstorage', f"{remove_ext}.png")
        os.remove(img_path)
        os.remove(f"{converted}/{remove_ext}.png")

#-- File cleanup
mypath = path
for root, dirs, files in os.walk(mypath):
    for file in files:
        os.remove(os.path.join(root, file))
```

Scoring and accuracy

Jaccard index

Jaccard index - Wikipedia

Clustering Comparisons

3 different clustering techniques were initially tested to determine how effective they are when compared against each other on the same data.

The 3 types tested are:

- K-Means
- DB Scan
- Gaussian Mixture Model (GMM)

The testing method was to generate a single cluster based on a test image. The generated cluster was then compared to a truth. Results were determined by:

1. Amount over identified data based on percentage of non flood region
2. Amount under identified data based on percentage of flood region

K-Means

Package:

Open CV 2

Settings:

```
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 5
attempts=10
cv2.kmeans(vectorized_fl,K,None,criteria,attempts,cv2.KMEANS_PP_CENTERS)
```

Results:

```
over identified 1.60% of non flood region
under identified 13.57 % of flood region
```

DBSCAN

Package:

Skilearn

Settings:

```
DBSCAN(eps=5, min_samples=500, n_jobs=-1)
```

Results:

```
over identified 0.36% of non flood region
under identified 21.23 % of flood region
```

Gaussian Mixture Model

Package:

Skilearn

Settings:

```
GaussianMixture(n_components=10)
```

Results:

```
over identified 0.53% of non flood region
under identified 15.69 % of flood region
```

Results: The comparisons between all 3 are close however GMM and KMeans have the better results. It was also noted that DBScan did take substantially longer to execute than GMM and Kmeans.

GMM is the more powerful of the 2 clustering packages. It has a much richer features and is highly configurable. KMeans is basic and less flexible over all. GMM has the ability to determine a points likely hood of belonging to a class where KMeans offers no such data. GMM also offers the ability to reuse a trained model for estimating other data sets based on the fitting already conducted giving the ability to reuse the model offering more predicability.

For these reasons we have chosen to pursue GMM as the chosen clustering method.

Testing and results can be found in the following notebook,

https://bitbucket.org/csu-spatialservices/ss-datascientist-platform/src/adam1/notebooks/2.2_CV_Adam_Clustering_Cont.ipynb?viewer=nbviewer

Image Pre Processing

Introduction

To ensure consistency across a batch of images, a study into image normalisation techniques was undertaken. A number of standard techniques was suggested including Normalisation, Centering, Standardising and Brightness and Contrast transformation. These techniques can be used independently or in conjunction where appropriate, and are useful on individual images or across a batch of images. For this project, processing images as a batch was considered to provide the most consistent result across all images, while selecting a single image as a standard and then basing the relative levels off that image is an option, a more automated solution is to average the existing values of all images and use these values as the standard to apply with the assumption that the majority of images are capable of being processed and only the minority are out of range.

Contrast and Brightness

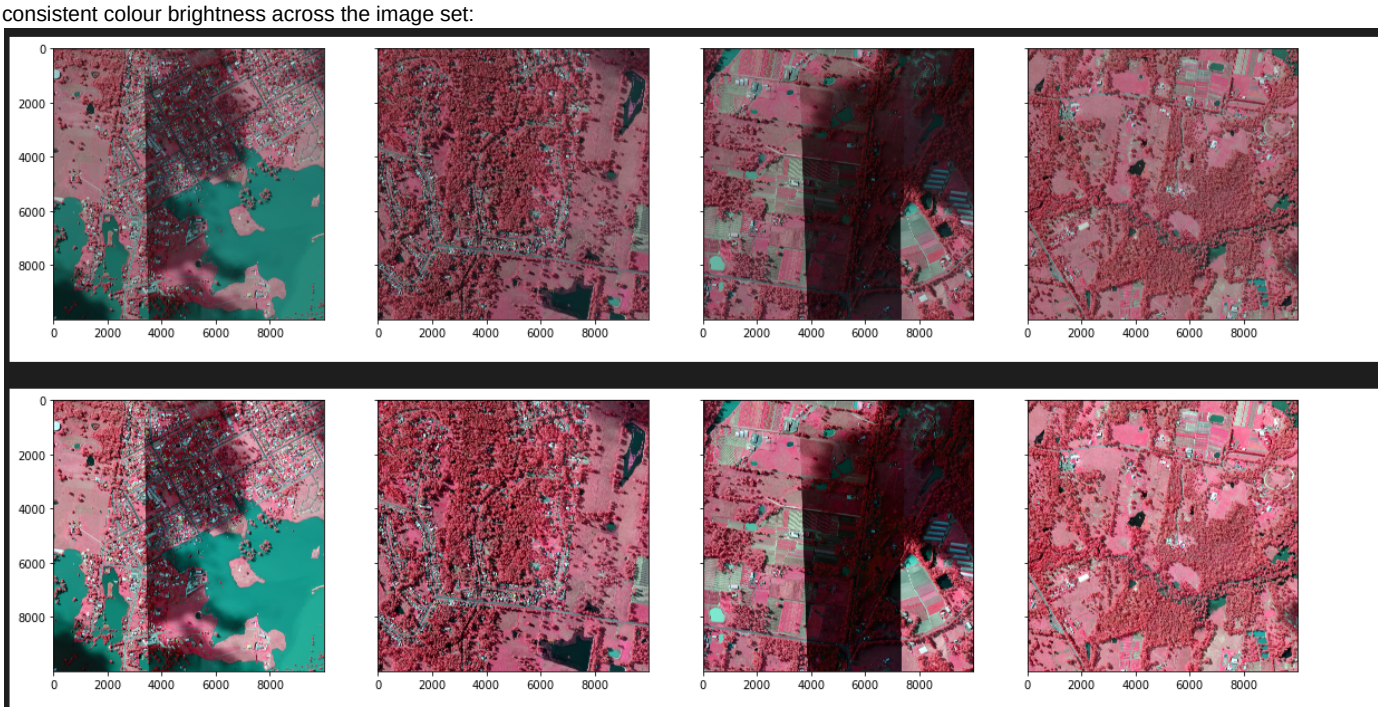
Associated notebook: https://bitbucket.org/csu-spatialservices/ss-datascientist-platform/src/darren1/notebooks/Auto%20Brightness%20Contrast.ipynb?viewer=nbviewer

The most effective solution was the Contrast and Brightness transformation which is used to adapt gain and bias of images using the equation g (x) = af(x) + b where g(x) is the output pixel, a is the contrast control, f(x) is the source image pixels and b is the brightness control. OpenCV provides a convenience method cv.convertScaleAbs(image, alpha=alpha, beta = beta) to save creating multiple loops to iterate through each pixel and apply the algorithm.

```
alpha = 1
beta = 0
result = cv2.convertScaleAbs(image, alpha=alpha, beta=beta)
```

The alpha and beta values are taken from each image and the average across all images is then used as the benchmark to apply alpha and beta values. To obtain the current alpha and beta values the image is converted to grayscale, a cumulative distribution is obtained from the histogram to obtain the lowest and highest percent of values to be clipped to reduce outliers, the alpha and beta values are then calculated and passed to cv2.convertScaleAbs with the original image to be processed.

The process to obtain the alpha and beta values over 4 images only takes less than 5 seconds, excluding image loading. The result is a more consistent colour brightness across the image set:



A variation of this method could involve selecting a single image as the benchmark and then base the brightness, contrast of the other images on that benchmark.

Normalized Pixel Values

In image processing normalizing pixels within an image refers to changing its colour range from an integer in the 0 - 255 range to a float value between 0 and 1. It is normal for pictures to be in the later range and this does not change how it is viewed. While initially it does not provide any benefit when converted, it does give the image the ability to have more variances with further processing, rather than being limited to 255 integers, there is now infinite numbers between 0 and 1 the pixels can use for their colour values.

To normalize an image in this way numpy provides a simple way of converting an image:

```
print("--Before--")
print("Image minumum level:",image.min())
print("Image maximum level:",image.max())

# convert to 0 - 1 range by dividing by 255
image = image.astype('float32')
image /= 255.0

print("--After--")
print("Image minumum level:",image.min())
print("Image maximum level:",image.max())
```

Centering Pixel Values

This technique uses the average of the pixel values, placing them at 0 by subtracting the mean from the pixel values, meaning that the pixel values range across negative and positive ranges. Doing this means that the images will not be viewable using normal image display, however machine learning cares more about the data than how it looks to a person.

Centering can happen at a global level where the mean across all channels in the image is taken and then used to center the image, or at the local level where the mean is calculated for each channel and then each channel is centered independently.

While we didn’t use this technique in the project, it’s worthwhile to be aware of its existince.

Image Standardisation

Pixel values are usually distributed over a normal bell curve meaning this can be used to reduce noise and outlier values to standadise image data. This can be done per image, or based on averages over a batch and done across all channels or on individual channels.

When combined with centering the bell curve centers on 0, and the standard deviation calcuated and clipped on either side of 0. To bring the images back into a viewable range between 0 and 1 the values can be recalculated as follows:

```
print("--Before--")
print("Image minumum level:",image.min())
print("Image maximum level:",image.max())
print("Image mean:",image.mean())
print("Image standard deviation:",image.std())

mean = image.mean()
sd = image.std()
image = (image-mean) / std
# clip values so they sit between -1 and 1
image = clip(image, -1.0, 1.0)
# move to positive
image = (image + 1.0)/2.0

print("--After--")
print("Image minumum level:",image.min())
print("Image maximum level:",image.max())
print("Image mean:",image.mean())
print("Image standard deviation:",image.std())
```

U-Net (Semantic Segmentation)

Following information and methods used by DigitalSreeni https://www.youtube.com/channel/UC34rW-HtPJulxr5wp2Xa04w.

Unet model as defined by here: https://github.com/bnsreenu/python_for_microscopists/tree/master/228_semantic_segmentation_of_aerial_imagery_using_unet

All testing done using python script from above repo.

1st round testing

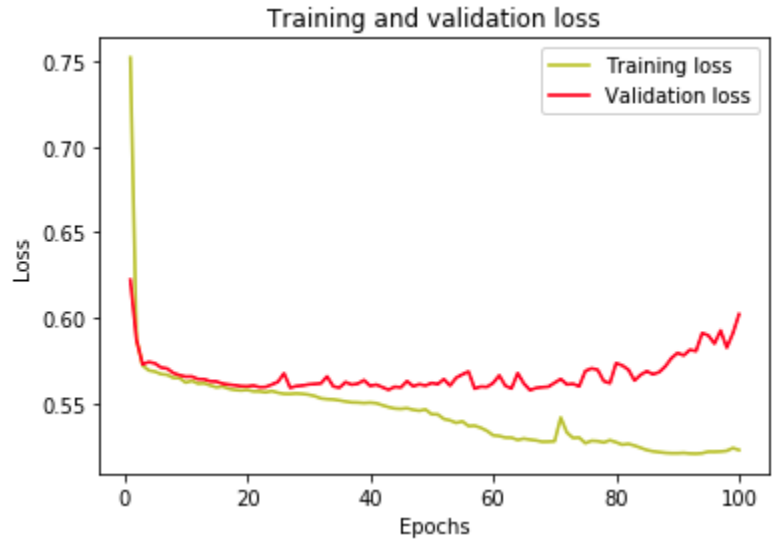
Was done using all 6 classes and and 100 epochs using the same script in the above repository. IOU scored ~0.6. However predict results suffered from large portions of pixels being identified as class of unlabelled. Weights were adjusted to combat this however this had little effect.

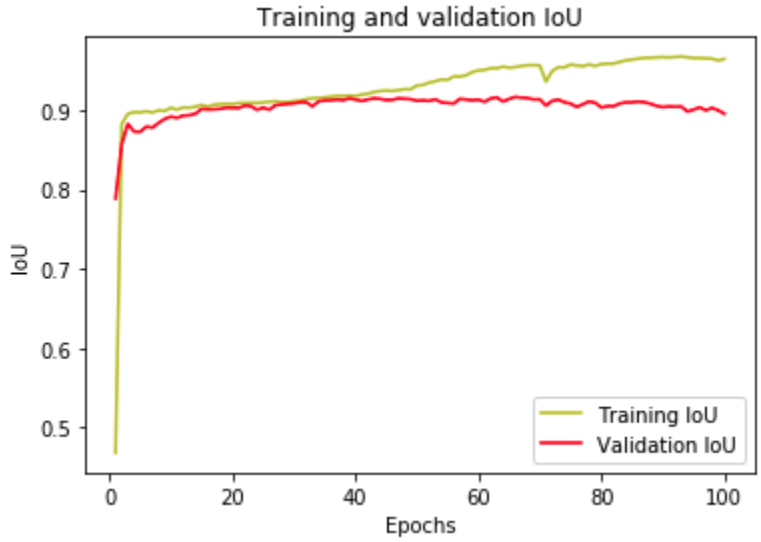
Data: Labelled Hawkesbury-Napean and Brewarrina images. **Classes:** unlabelled, flood, flood + cloud, building, trees, land. weights: (0.1666, 0.1666, 0.1666, 0.1666, 0.1666, 0.1666)

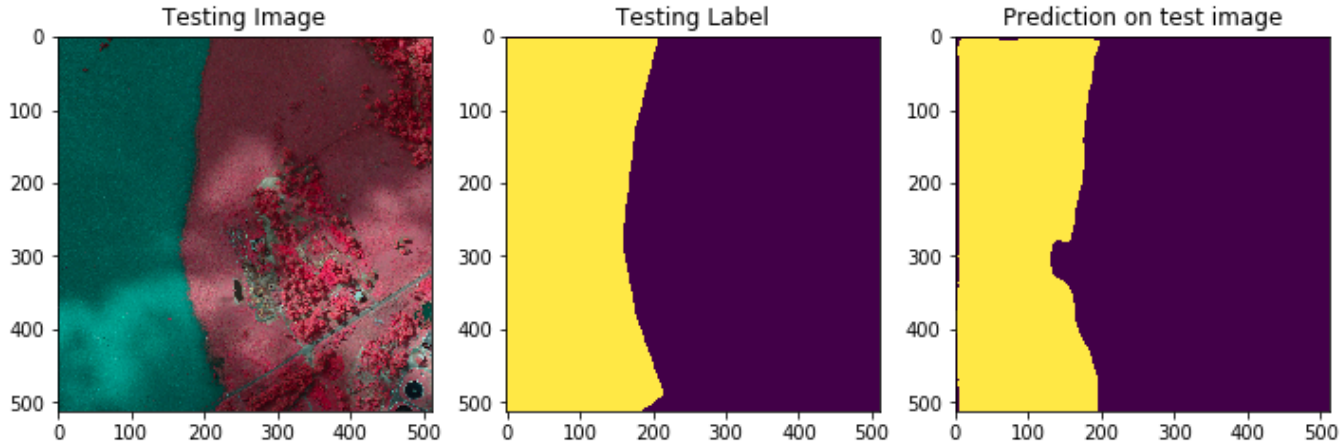
2nd round testing

Limited classes by combing all flood into one flood class and others into one non flood class. IOU training scored 0.9, and 100 epochs.

Data: Labelled Hawkesbury-Napean and Brewarrina images. **Classes:** flood, non flood. weights: (0.5, 0.5)





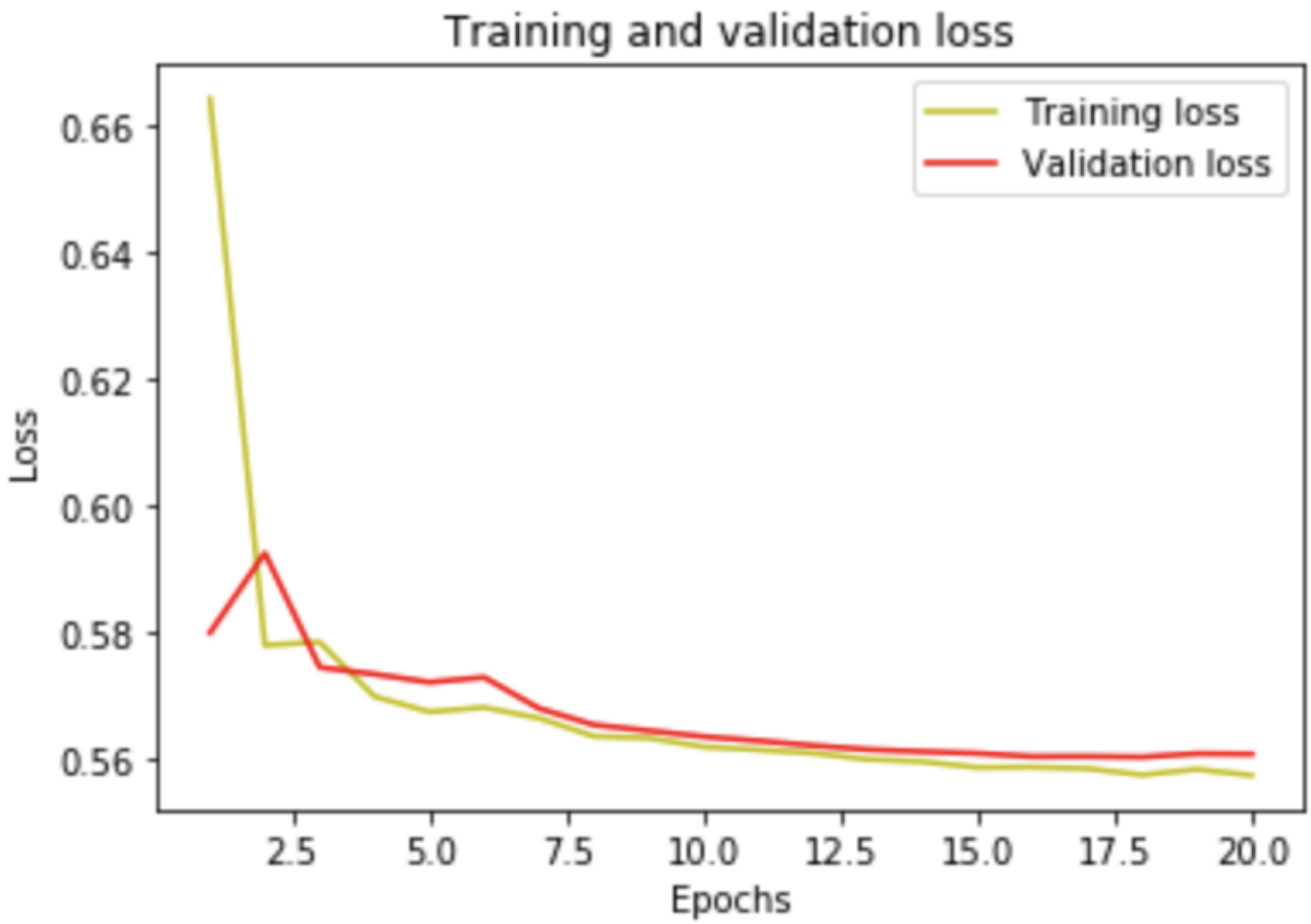


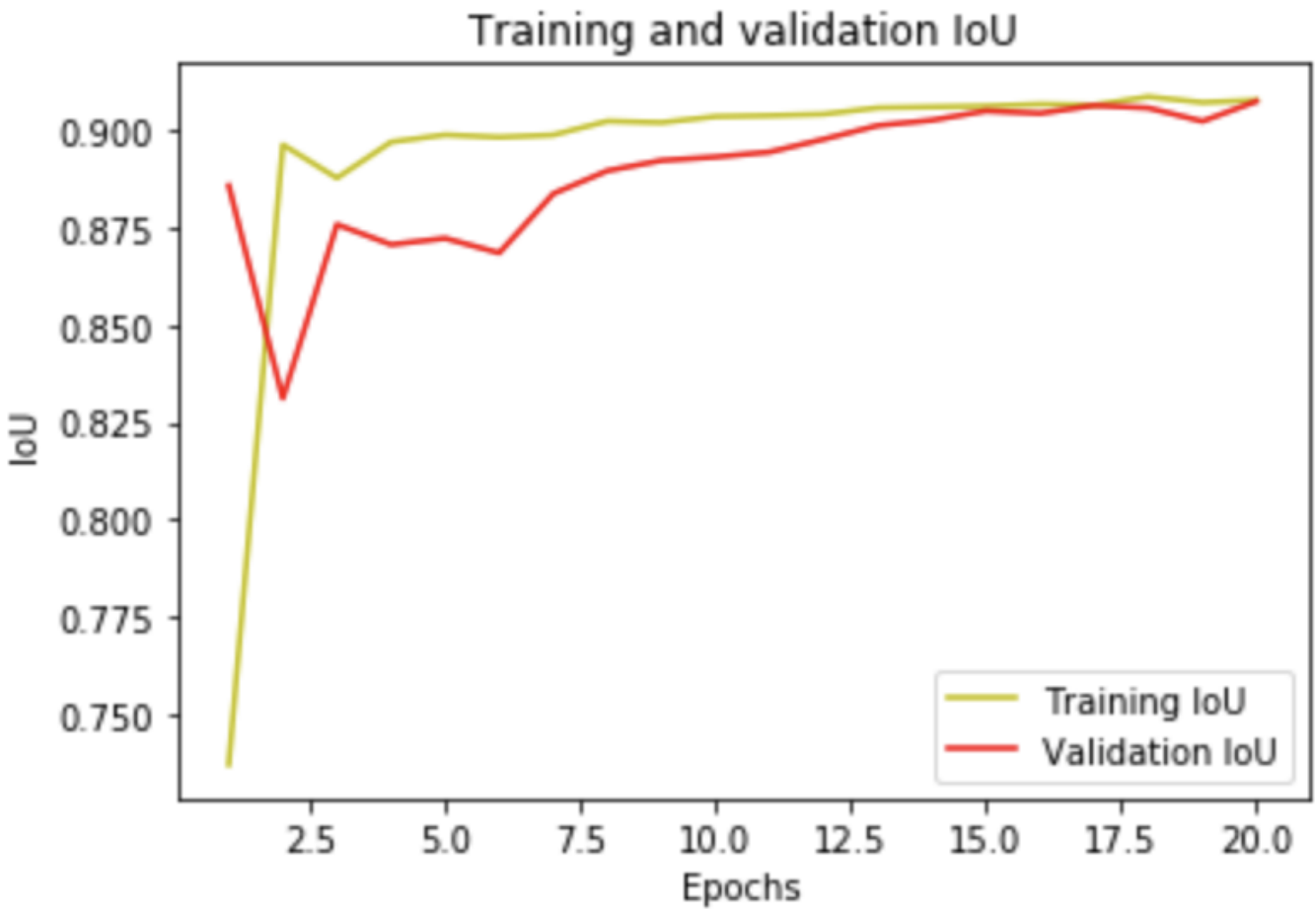
Limiting class to flood and non-flood increased IOU scores dramatically. Looking at IOU and loss graphs indicates model is not improving past ~20 epochs. Reducing epochs may result in better segmentation.

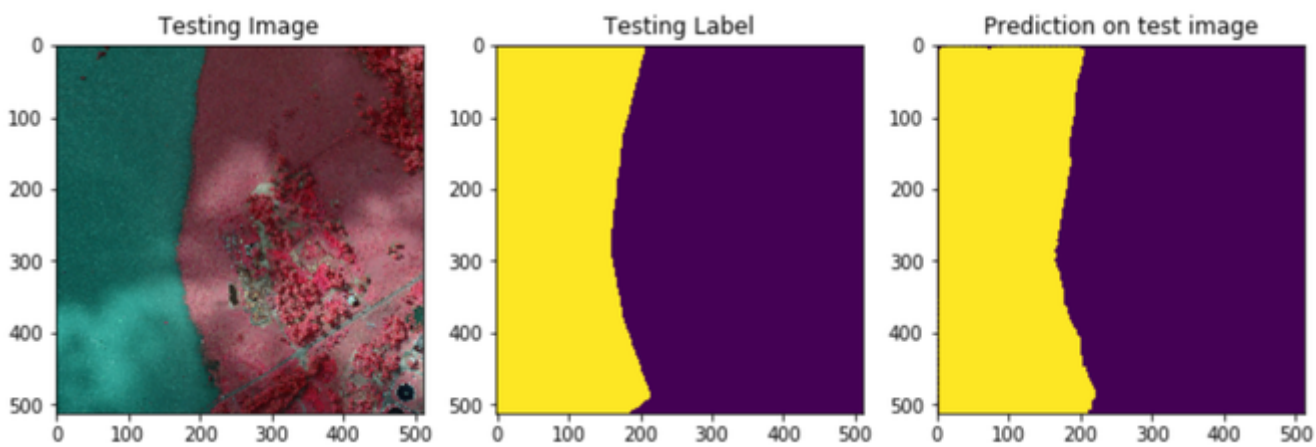
3rd round testing

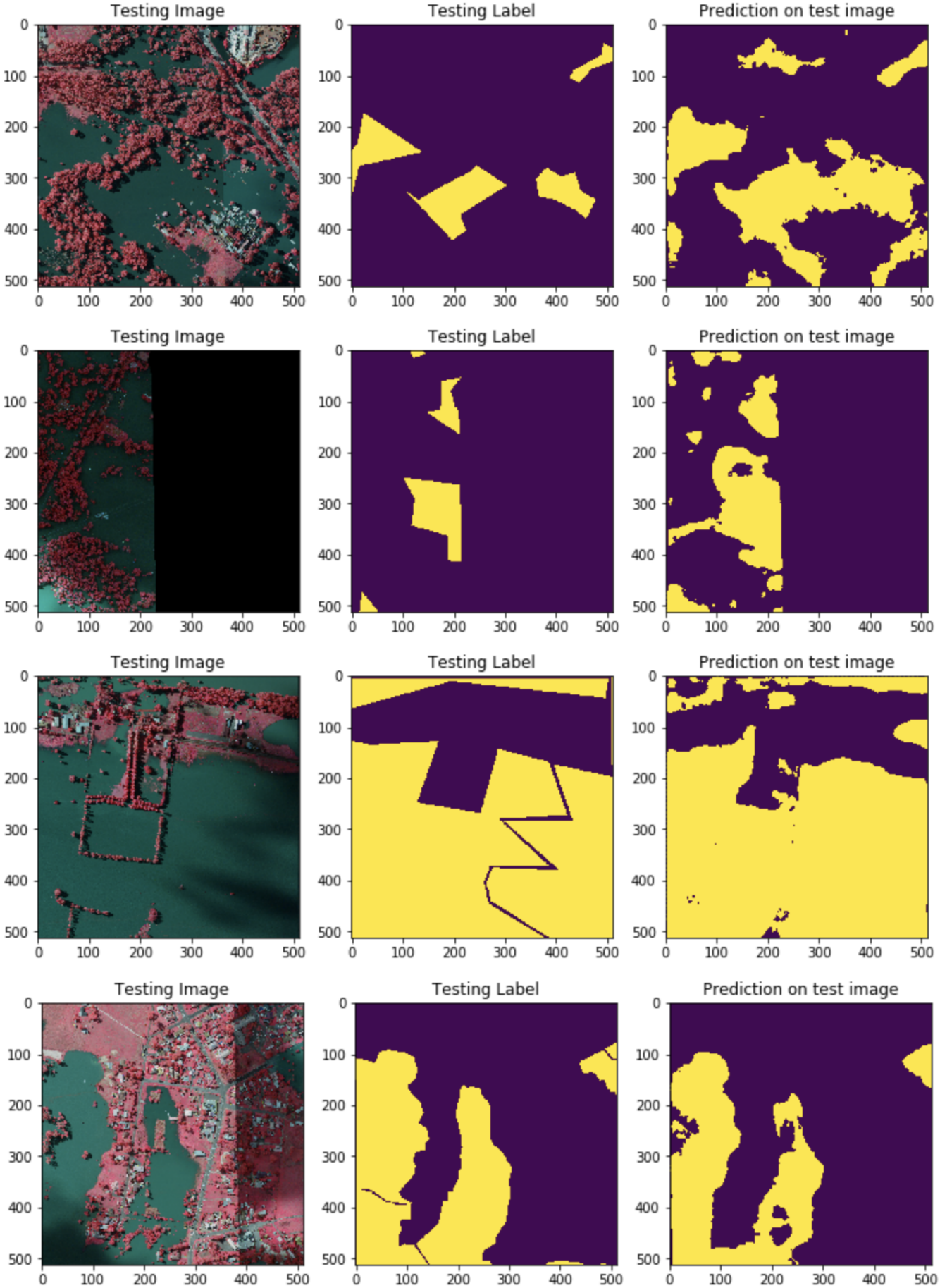
Reducing epochs to 20 to reduce effects of over training. Results in similar IOU as 2nd round testing but produces more visually accurate predictions.

Data: Labelled Hawkesbury-Napean and Brewarrina images. **Classes:** flood, non flood. weights: (0.5, 0.5)




Training and validation IoU
IoU
0.900
0.875
0.850
0.825
0.800
0.775
0.750
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epochs
Training IoU
Validation IoU

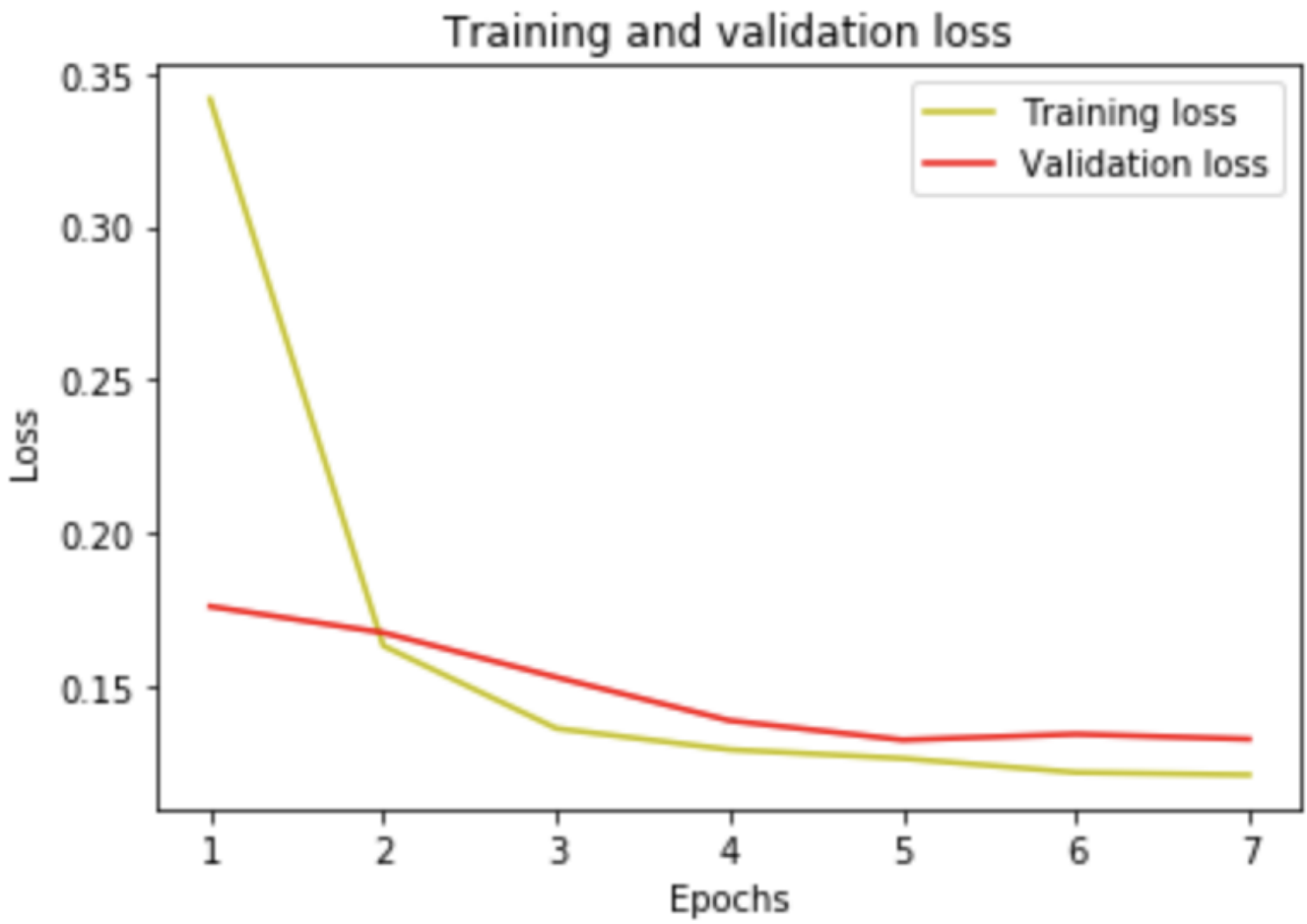

Testing Image
Testing Label
Prediction on test image
0
100
200
300
400
500

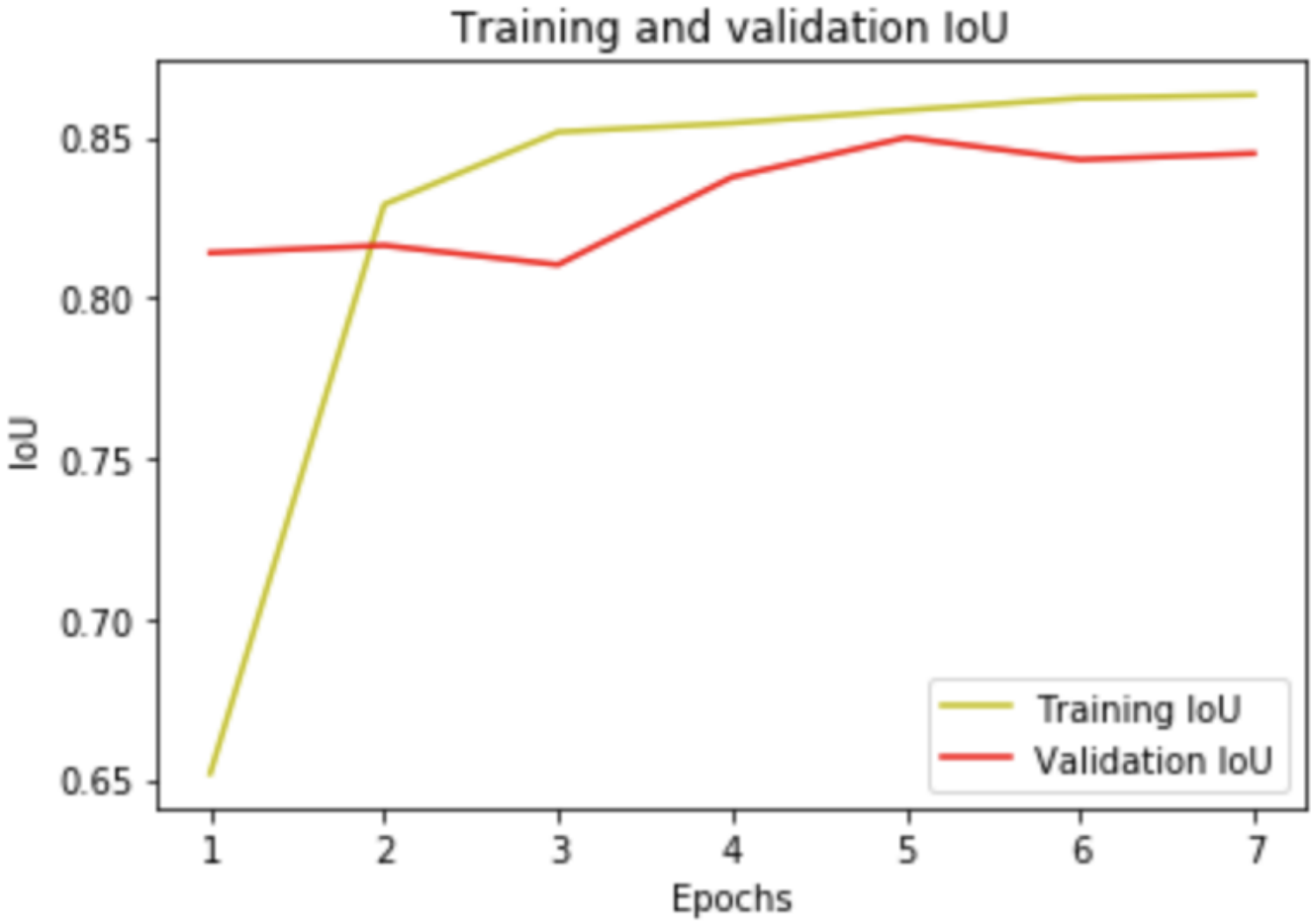

Testing Image
Testing Label
Prediction on test image
0
100
200
300
400
500

Current training / fitting is using custom lose function and multi class segmentation. As this is a binary classification problem now (2 classes) a binary lose function should be tested.

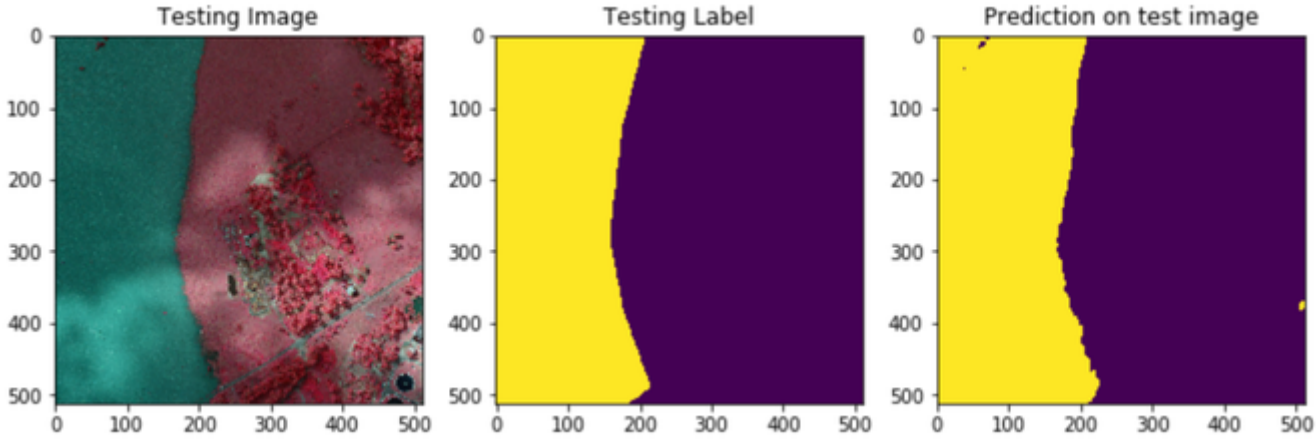
4th round testing

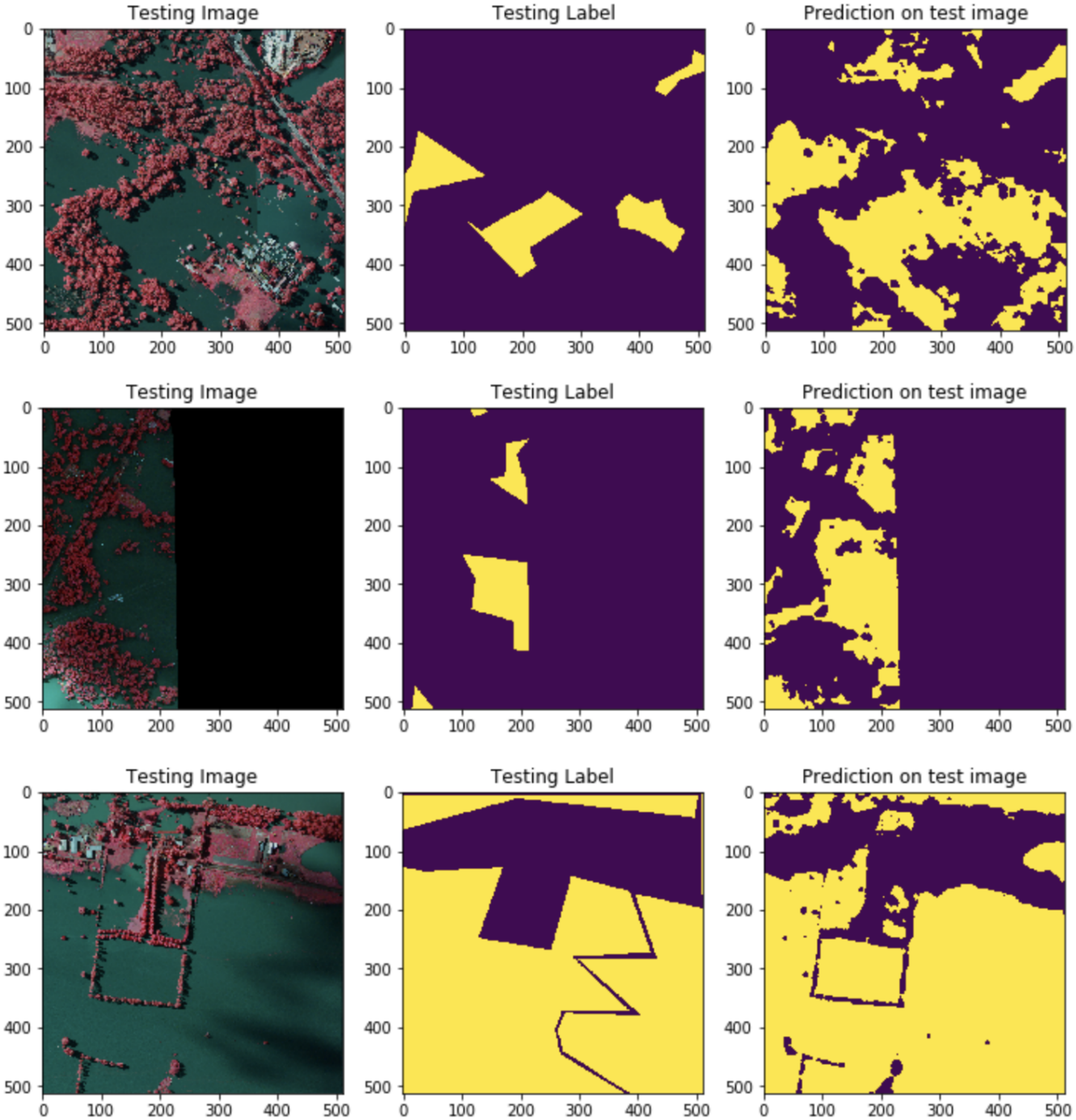
Model fitting was modified to use binary_crossentropy as the loss function. Fitting method also now exits early based loss_val. Model was able to exit after 7 epochs (2 used for patience, so 5 epochs) with a final IoU of .85.

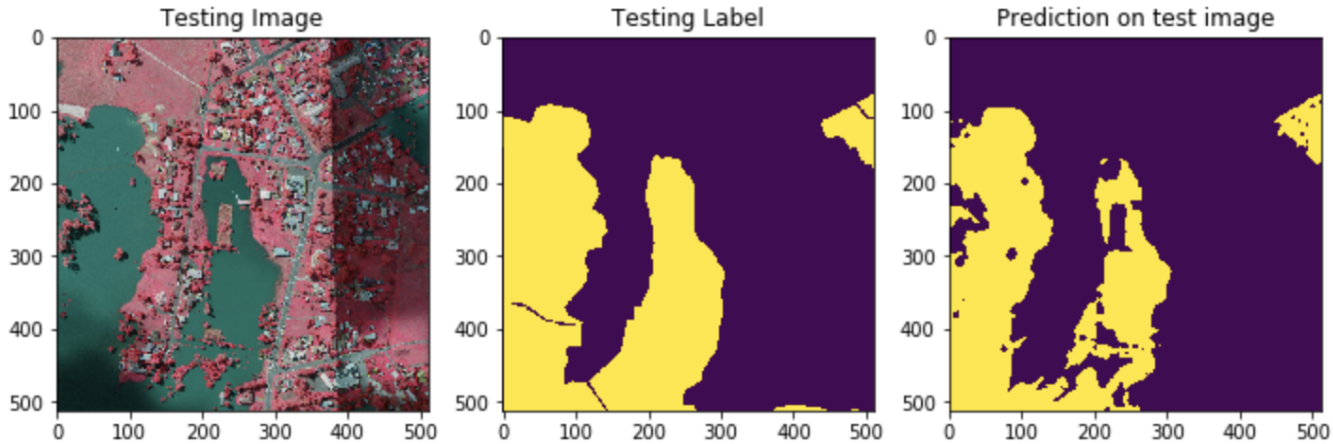



Training and validation IoU
0.85
0.80
0.75
0.70
0.65
IoU
1
2
3
4
5
6
7
Epochs
Training IoU
Validation IoU

```
callbacks = [
        tf.keras.callbacks.EarlyStopping(patience=2,
monitor='val_loss'),
        tf.keras.callbacks.TensorBoard(log_dir='logs')]

from simple_multi_unet_model import multi_unet_model, jacard_coef

metrics=['accuracy', jacard_coef]

def get_model():
    return multi_unet_model(n_classes=n_classes, IMG_HEIGHT=IMG_HEIGHT,
IMG_WIDTH=IMG_WIDTH, IMG_CHANNELS=IMG_CHANNELS)

model = get_model()
model.compile(optimizer='adam', metrics=metrics,
loss='binary_crossentropy')
model.summary()

history1 = model.fit(X_train, y_train,
                     batch_size = 16,
                     verbose=1,
                     epochs=100,
                     validation_data=(X_test, y_test),
                     shuffle=False,
                     callbacks=callbacks)
```




Testing Image
Testing Label
Prediction on test image
Testing Image
Testing Label
Prediction on test image
Testing Image
Testing Label
Prediction on test image
0
100
200
300
400
500



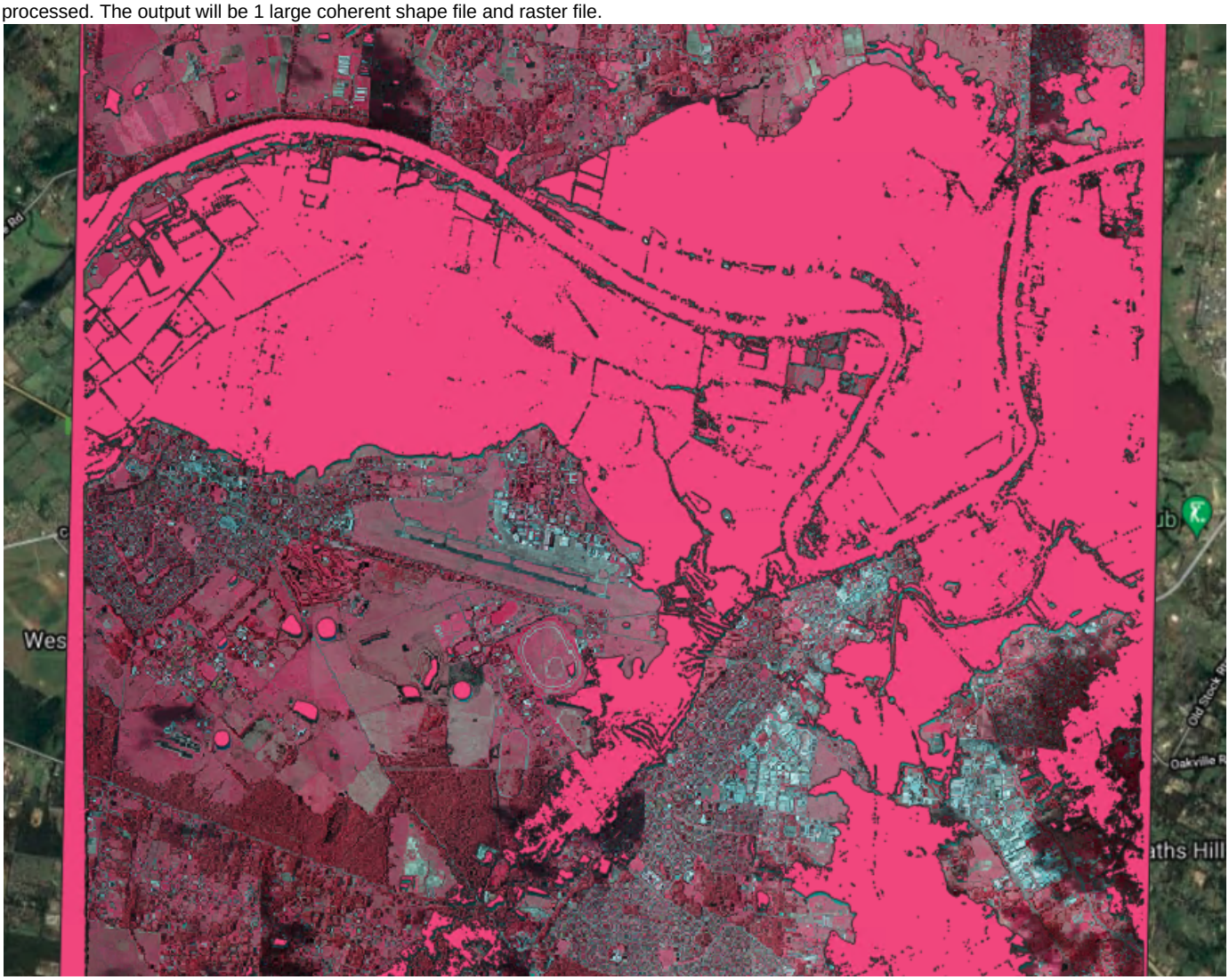
Segmented flood are is now much more detail than previous 3rd round results. Next step would be to create a custom Unet model that can handle images in their native size. Currently image are rescaled to 11000px and down sampled to 1100px before being cropped to 1024px. This is to cope with the current Unet model.

Over all the results are robust and further development could yield better results.

Combining Images / ShapeFiles

Often it is not practical or possible to process single large datasets / images.

We have devised a method that allows datasets that have been broken into smaller images (aka tiles) before being given to the Notebook to be processed. The output will be 1 large coherent shape file and raster file.



High overview of the steps:

1. As each image is processed the output is a binary image of the flood area (0 = Non flood, 1= flood)
2. The original image/s are opened as a dataset using rasterio. The dataset contains the same geodetic information as the source image but with no pixel / raster data.
3. We add the binary image as a raster layer to the dataset opened in step 2.
4. Once this has been completed for all images we merge all the datasets to create on single large data set. This single dataset now contains all the flood data correctly positioned. This single dataset is saved as a JP2 image.
5. Contouring / Polygon creation is performed on the one large single image
6. The shape is then generated from the polygons found in step 5

```python
# 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)
```

This method is utilised in both GMM and UNet Notebooks

https://bitbucket.org/csu-spatialservices/flood-extent-extraction/src/main/App/GMM%20Extraction%20Method/GMM_Flood_Extent_Extraction.ipynb?viewer=nbviewer

https://bitbucket.org/csu-spatialservices/flood-extent-extraction/src/main/App/Unet%20Extraction%20Method/Unet_Flood_Extent_Extraction.ipynb?viewer=nbviewer

Image annotation

Okay so we can start the image annotation. Using https://www.robots.ox.ac.uk/~vgg/software/via/via-1.0.6.html

STEPS:

1. Load 1 image

2. Annotate using polygons (or otherwise if required)

3. Create an column under region attributes called element (as seen in the image below)

VGG Image Annotator

VIA is a standalone image annotator application packaged as a single HTML file (< 200 KB) that runs on most modern web browsers.

4. Label items and regions under the following categories - Flood - Flood + cloud - Building - Trees - Land

5. Rename the JSON file to <imagename>_json

6. Upload to s3

Knowledgebase

This is a section where common knowledge can be shared, regarding configuration, setup and standardisation of formatting conventions.

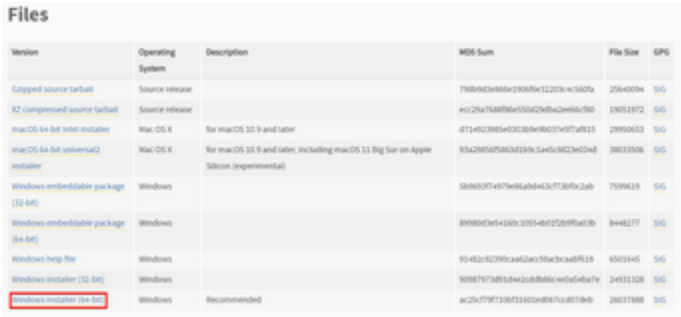
Configuring AWS CLI/Jupyter for Windows

***Disclaimer: I'll be running through this on Python 3.9.6**

Dependencies

Python 3.9.6

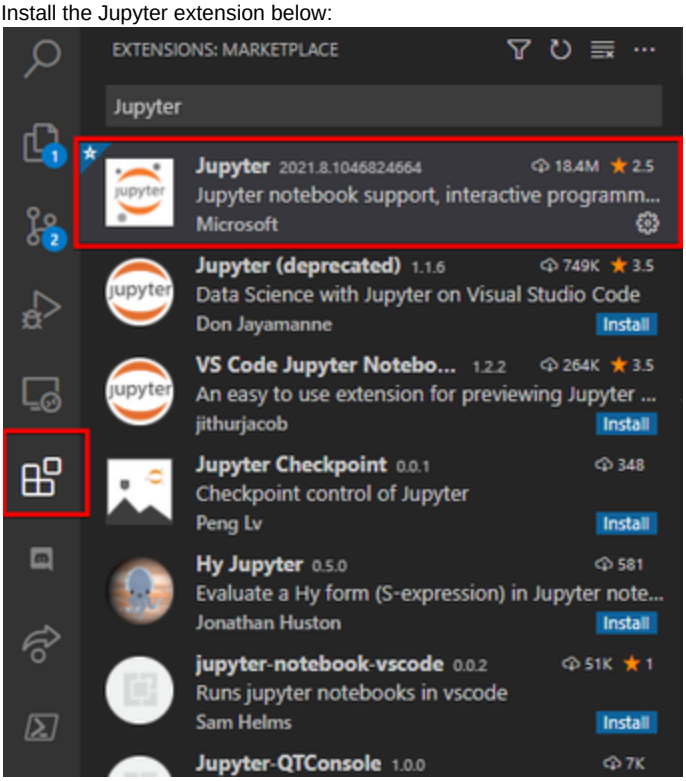
https://www.python.org/downloads/release/python-396/



Visual Studio Code

https://code.visualstudio.com/

Install the Jupyter extension below:



Python Dependencies

Open a Command Prompt (as Administrator),

Installing Python/Notebook/CV libraries:

```
pip3 install ipykernel
pip3 install numpy
pip3 install pandas
pip3 install opencv-python
pip3 install matplotlib
pip3 intsall scikit-image
pip3 install sklearn
pip3 install pickle-mixin
```

Installing AWS libraries:

```
pip3 install aws
pip3 install aws-mfa
pip3 install boto3
pip3 install botocore
```

Visit both of these links:

Download the latest version of each with the cp that matches your Python version (e.g. cp39 = Python 3.9.x)

https://www.lfd.uci.edu/~gohlke/pythonlibs/#rasterio

https://www.lfd.uci.edu/~gohlke/pythonlibs/#gdal

Open a Command Prompt (as Administrator),

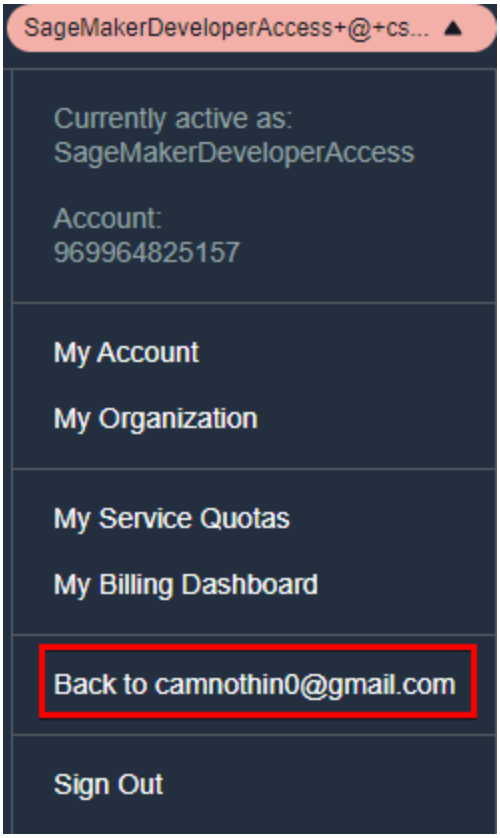
(Press the buttons between the square brackets when prompted)

```
cd %UserProfile%\Downloads

pip3 install wheel
pip3 install pipwin

pip3 install gdal [TAB] [ENTER]
pip3 install rasterio [TAB] [ENTER]

pipwin install fiona

pip3 install earthpy
```

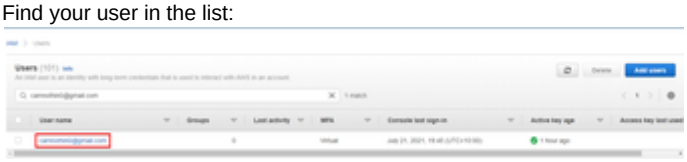
Configuring AWS CLI

Visit IAM in AWS - https://console.aws.amazon.com/iamv2/home#/users

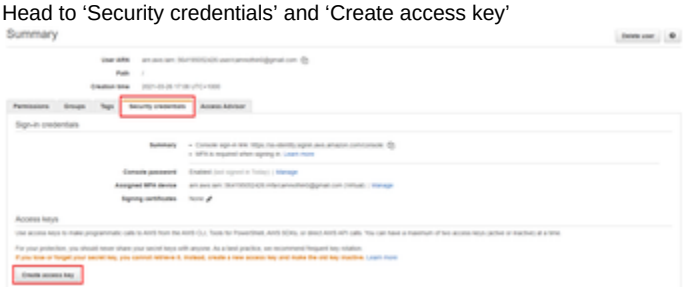
Ensure you're logged in as your user (not under the SageMakerDeveloperAccess role)



Find your user in the list:

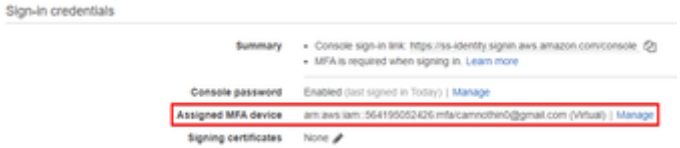


Head to ‘Security credentials’ and ‘Create access key’



Note down the ‘Access key ID’ and the ‘Secret access key’, then hit ‘Close’.

Also note down your ‘Assigned MFA device’ and **NOT** the ‘User ARN’ above.



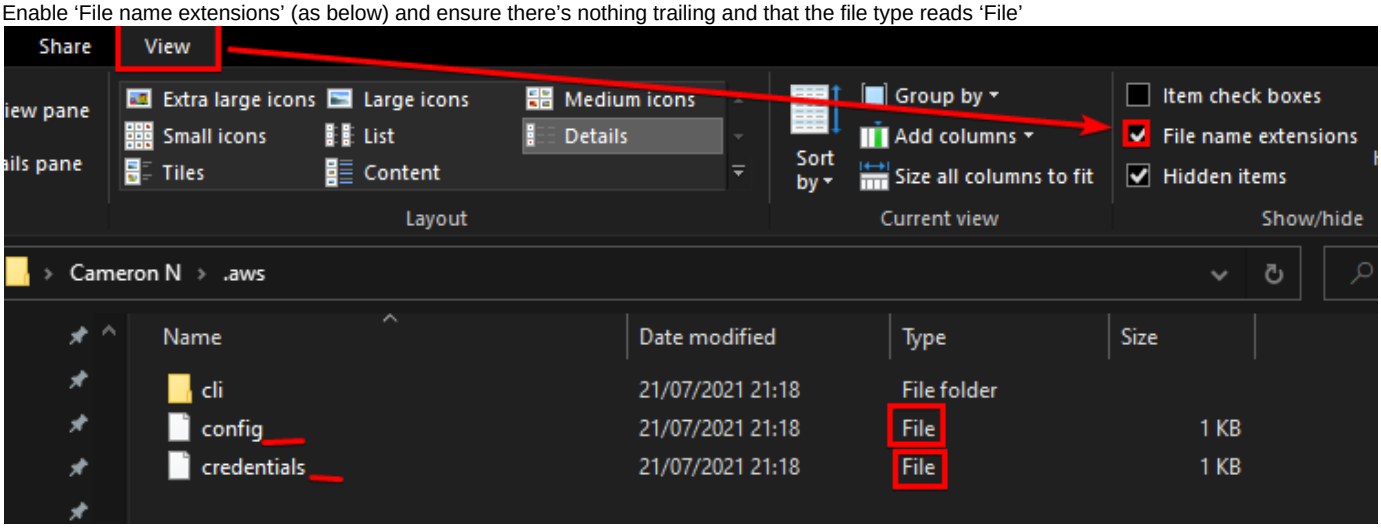
Open a new File Explorer window and navigate to

> *%UserProfile%*

Create the folder (if it doesn’t exist) ‘.aws’

Inside the .aws folder, create a file ‘config’ and another ‘credentials’

Enable ‘File name extensions’ (as below) and ensure there’s nothing trailing and that the file type reads ‘File’



Open credentials (in Notepad) and configure the following:

> *[default-long-term]*
> *aws_access_key_id = YOUR_ACCESS_KEY_ID*
> *aws_secret_access_key = YOUR_SECRET_ACCESS_KEY*
> *aws_mfa_device = YOUR_ASSIGNED_MFA_DEVICE*

The MFA device will start with arn: and end with your username.

Open config (in Notepad) and configure the following:

> *[profile default]*
> *region = ap-southeast-2*
>
> *[profile csu]*
> *source_profile = default*
> *role_arn = arn:aws:iam::969964825157:role/SageMakerDeveloperAccess*

Test

Open a Command Prompt (as Administrator),

```
aws-mfa --duration <time in seconds before you need to retrieve another mfa code>
```

The screen should read Success! if all is correct!

Then attempt the following:

```
aws s3api list-buckets --profile csu
```

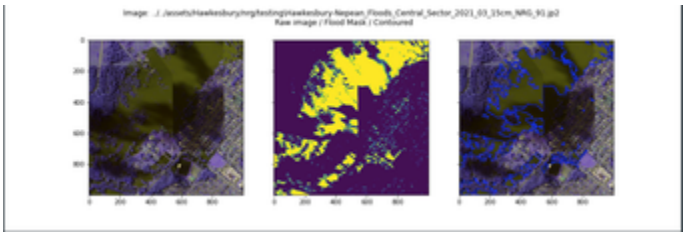
And you should now have a list of all buckets you have access to!

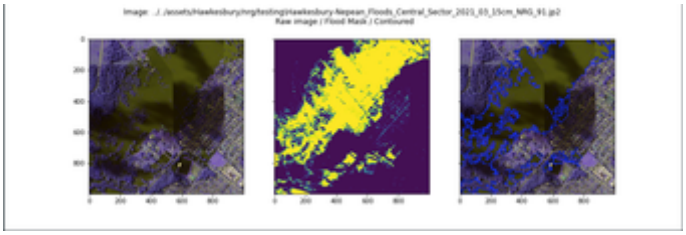
Noting we specify ‘--profile csu', as this is the name of the ‘profile’ in our config file, that we gave the SageMakerDeveloperAccess role to.

Image resizing

During testing I discovered that images resized with OpenCV 2 perform differently in GMM clustering than images resized with skimage rescale. This is most likely due to interpolation techniques used by each type.

Care should be taken to ensure consistent use of resizing for both creation and testing of models.



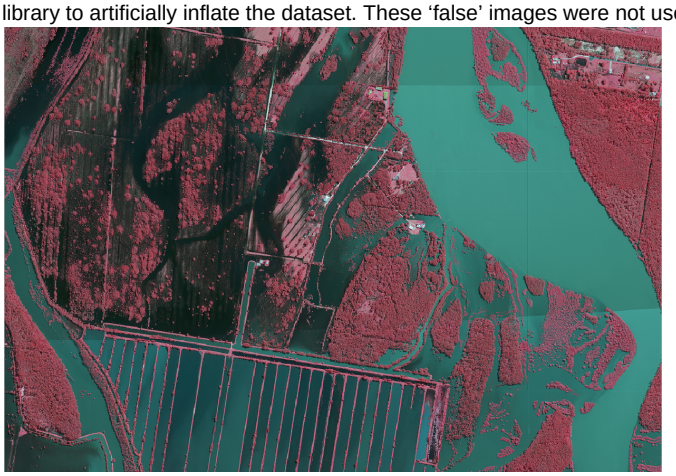


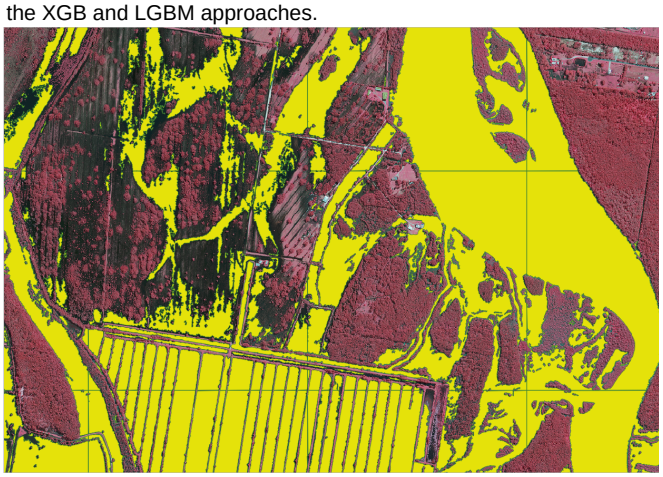
Supervised Semantic Segmentation

In the field of computer vision, semantic segmentation classifies each pixel within an image to one of a number of classes. It can be implemented through both supervised and unsupervised machine learning techniques. This section provides an introduction to the core concepts in supervised machine learning and then covers the approaches implemented through the AWS SageMaker API for Fully-Convolutional Neural Networks, Pyramid-Parsing Scheme, Deeplab 3.0, as well as scikit-learn implementations of Extreme Gradient Boosting (XGB) Trees, as well as its combination with the unsupervised Gaussian Mixture Model (GMM) approach.

Hyperparameter tuning and artificial inflation of training datasets are also covered in the below sections. Hyperparameter tuning provides a way to optimise parameters used in objective functions and learning algorithms as discussed in the Introduction to Supervised Machine Learning section.

All implementations of semantic segmentation utilised by G5 use the same data for training and validation sets, i.e. the Hawkesbury-Nepean and Brewarrina datasets. For the AWS SageMaker algorithms, these datasets were supplemented with augmented images using the imgaug Python library to artificially inflate the dataset. These 'false' images were not used in the XGB and LGBM approaches.



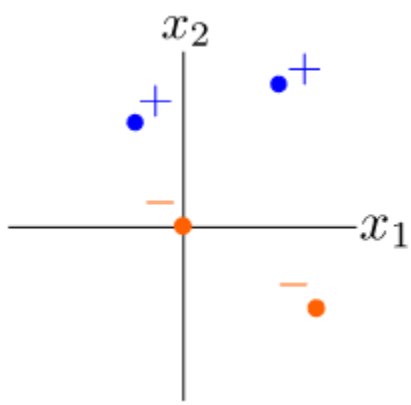


Introduction to Supervised Machine Learning

Machine learning aims to design programs that learn from experience for the purposes of modelling, prediction, or control. While there are numerous types of machine learning tasks, this section will focus on supervised machine learning in which predictions are made based on examples of past behavior. A common use case of supervised learning are recommender programs that suggest films and TV shows to viewers on platforms such as Netflix. Based on a viewer's history of watched films and given ratings, recommender programs aim to predict which unseen films and TV shows a viewer will also enjoy. Trying to solve such problems directly by specifying some set of rules is prohibitively difficult. Supervised machine learning instead aims to automate the process of finding a solution by learning from examples. This involves hypothesizing a set of parameterized functions, or mappings, that describe the relationship between the inputs (viewer watch-list) with the outputs (prediction of TV shows, films the user will also enjoy). This reformulates the problem into an optimization task where the parameters that most accurately describe the relationship between the inputs and the outputs are honed through iterative learning algorithms. If there exists a labelled data-set, the accuracy of the parameters used can be verified by dividing the data into two subsets; one for training the learning algorithm which hones the values of the parameters; the other for testing the accuracy of the parameters by looking at the margin between predicted outputs and actual outputs. The specific type of functions or mappings that are used to describe the relationship between the inputs and outputs correlate to different algorithmic implementations of supervised learning such as Support Vector Machines (SVMs), Extreme Gradient Boosted Trees (XGBoost) or Convolutional Neural Networks (CNNs).

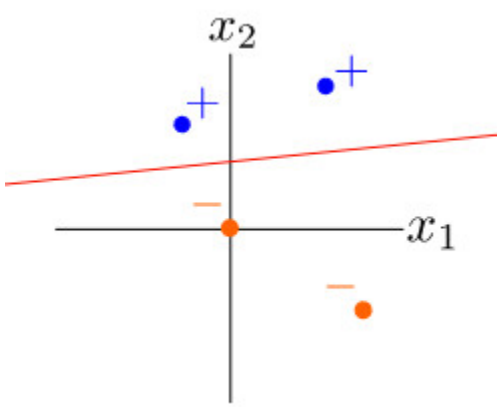
It should be noted that AWS SageMaker provides API implementations for all algorithms discussed here, allowing users to treat these as a 'black box' without understanding the underlying structure. For the purposes of understanding the overall problem, a high-level overview is given below.

The core concepts behind supervised learning tasks can be illustrated with a simple data-set where the goal is to make a binary classification. In the context of the recommender program the classification could be whether or not to recommend some film to a particular viewer. For simplicity, assume the films can be represented by two variables, meaning each film has a two-dimensional feature vector. Further assume that the data-set is linearly separable, i.e. a line can be drawn through the data-set on a graph and on one side of the line will lie all the movies a user will enjoy, and on the other will lie all the movies that same user will not enjoy. This line is known as the decision boundary.



A plane of any dimension *n-1* can be represented by $+ {}_0$ where and are of dimension *n*. In the above example $= [{}_1, {}_2]^T$ is the two-dimensional feature vector describing the film, $= [{}_1, {}_2]^T$ is the parameter to be honed by the learning algorithm and ${}_0$ is some scalar offset to be similarly adjusted. The decision boundary will be a straight line bisecting the data. If the data-set were three-dimensional, the decision boundary would be a plane of two-dimensions. A plane of 3 dimensions or higher is called a hyperplane. Note that if there were no ${}_0$ term this would imply the decision boundary would pass through the origin which is not applicable in the example given.

Clearly, there exists at least one line (in fact infinitely many) that can correctly separate the data in Figure 1. This line would be represented by $+ {}_0 = 0$ and correlates to the red line in Figure 2. This line would be found by adjusting the values for and ${}_0$ through the use of some learning algorithm.



When substituting in the the values for correlating to the two positively labeled data points into the equation $+ {}_0$, the output will be > 0 and those points will lie above the decision boundary in Figure 2. Performing the same operation for the negatively labelled data-points will produce values < 0 and likewise those points will lie below the decision boundary in Figure 2. Hence, the predictions ${}_i$ are the output of ${}_i + {}_0$.

Given that there are infinitely many decision boundaries to select from, how should an optimal one be selected? This is done through the use of an objective function which has a loss term and a regularization term. The decision boundary with parameters and ${}_0$ that minimize this function is the one that ends up being selected. The loss term measures the accuracy of the decision boundary. It provides a numerical value representing the degree to which data-points in the testing subset are accurately classified. The regularization term provides a numerical value that penalizes large values of and ${}_0$. The mathematical intuition is to prevent the learning algorithm from finding a decision boundary that too closely fits the training data. If this regularization term were ignored, the resulting decision boundary would be excellent at accurately classifying the training data, but would likely be terrible at classifying any unseen data that slightly deviates from the training data. This is known as overfitting. The decision boundary is thus a result of finding values of and ${}_0$ found from the learning algorithm that also minimize the objective function.

$$\mathrm{obj}(\theta) = L(\theta) + \Omega(\theta)$$

One learning algorithm that could be used to arrive at appropriate values of and ${}_0$ for a linearly separable data-set is the perceptron algorithm. A discussion of various learning algorithms is beyond the scope of this section. The above example deals with a linearly-separable two-dimensional data-set but a similar process can be applied to data-sets of higher dimensions which are also non-linearly separable. The decision boundary could have various implementations from complex polynomials to decision tree ensembles which may also be mapped into a higher dimensional feature space to find an appropriate boundary in the data which is in turn mapped back into the original feature space. Again, a discussion on various representations of decision boundaries is beyond the scope of this section. The main points to learn are:

- a supervised machine learning task learns how to model the relationship between inputs and outputs (here some prediction) by passing existing examples through a learning algorithm
- the learning algorithm optimizes the values of and ${}_0$ which determine the decision boundary
- the set of potential decision boundaries are evaluated through an objective function which measures the accuracy and complexity of the values of and ${}_0$
- the goal is to find a decision boundary that is both accurate on the training data as well as on unseen new data
- values of and ${}_0$ that minimize the objective function are taken to satisfy these constraints
- the concepts here represented on low-dimension datasets can be extrapolated to arbitrarily-large dimension datasets that are non-linearly separable

Data Preparation

The SageMaker semantic segmentation algorithm provides a pixel-level approach to computer vision applications. Like any supervised machine learning task it requires a labelled data-set in order to learn the mappings from the input data to the predicted labels. The input data in this case are the red, near-infrared, and green channels for each pixel. The labels are a set of classes chosen by G5 under the guidance of Intellify. The labels are:

- flood
- flood + cloud
- buildings
- trees
- land

In the context of the equation introduced in Introduction to Supervised Machine Learning, the labels y_i = [flood, flood + cloud, buildings, trees, land] and the inputs x_{ij} = [red, near-infrared, green] for each pixel in each image.

Labeling of images from the Hawkesbury-Nepean and Brewarrina flood data-sets was performed by the student team using the VGG Image Annotator, an online open-source image annotation service. Students uploaded the input images into the tool and manually drew polygons over objects of interest in the images. The output of this process was a JSON file containing a list of polygons, their labels, and Cartesian coordinates.

A Python script was written by the team to translate this data into an image containing the annotated polygons. Depending on their label, the polygons were given a grayscale color. The labels are thus associated with a given color intensity. This is a requirement for the training process.

LABEL	GRAYSCALE VALUE
flood	50
flood + cloud	100
buildings	150
trees	200
land	250

The AWS SageMaker Semantic Segmentation model requires input data in a specific format. First, the data must be split into train and validation subsets. Second, for every train and validation image, there is a corresponding annotation image which provides a pixel-wise label. This provides the machine learning model with the necessary information to 'learn' which pixels represent floods, floods with cloud-cover, trees, buildings, and land.

There will be in total four folders of images:

- `train`
- `train_annotation`
- `validation`

- `validation_annotation`

In addition to the above folders there may be an additional folder: `label_map`. This folder contains two JSON files; `train_label_map.json` & `validation_label_map.json`. These provide an explicit mapping of grayscale values to labels. This is necessary since the semantic segmentation does not accept explicit semantically meaningful labels such as 'flood' or 'trees' as input. Rather, each label is labelled as an index. For a data-set with n labels, the range of total label indices is 0, 1, 2,...,n-1. For the current data-set the indices range from 0 to 4. The mapping of grayscale values to label indices is provided below.

Below is the script used to convert JSON files to annotated PNG images. An online tutorial was used to assist the development of this script. https://towardsdatascience.com/generating-image-segmentation-masks-the-easy-way-dd4d3656dbd1

```
import os
import cv2
import json
import numpy as np
from PIL import Image

json_path = "json_annotations/{}"
output_path = "json_annotations/png_gray_annotations/"
png_path = "/home/andrew/Documents/flood_images/png/"
os.makedirs(output_path, exist_ok=True)

flood_counter = 0
flood_cloud_counter = 0

colors = {
        "flood": (50),
        "flood + cloud": (100),
        "building": (150),
        "buildings": (150),
        "trees": (200),
        "land": (250)
}


def extract_region(data, itr, key):
        """Extract x/y coords and label from polygon in json file."""
        if type(iter(data[itr]["regions"])).__name__ ==
"dict_keyiterator":
                key = str(key)
        shape = data[itr]["regions"][key]["shape_attributes"]["name"]
        if (shape == "polygon"):
                try:
                        x_points = data[itr]["regions"][key]
["shape_attributes"]["all_points_x"]
                        y_points = data[itr]["regions"][key]
["shape_attributes"]["all_points_y"]
                        label = data[itr]["regions"][key]
["region_attributes"]["element"].strip().lower()
                except:
                        print("Could not extract polygon vector points.
Skipping", key)
                        return
```

```
                all_points = []
                for i, x in enumerate(x_points):
                        all_points.append([x, y_points[i]])

                return [all_points, shape, label]
        elif (shape == "rect"):
                try:
                        x1 = data[itr]["regions"][key]
["shape_attributes"]["x"]
                        y1 = data[itr]["regions"][key]
["shape_attributes"]["y"]
                        width = data[itr]["regions"][key]
["shape_attributes"]["width"]
                        height = data[itr]["regions"][key]
["shape_attributes"]["height"]
                        label = data[itr]["regions"][key]
["region_attributes"]["element"].strip().lower()
                        all_points = [x1, y1, x1+width, y1+height]
                except:
                        print("Could not extract rectangle coordinates.
Skipping", key)
                        return
                return [all_points, shape, label]
        elif (shape == "circle"):
                try:
                        cx = data[itr]["regions"][key]
["shape_attributes"]["cx"]
                        cy = data[itr]["regions"][key]
["shape_attributes"]["cy"]
                        radius = data[itr]["regions"][key]
["shape_attributes"]["r"]
                        label = data[itr]["regions"][key]
["region_attributes"]["element"].strip().lower()
                        all_points = [cx, cy, radius]
                except:
                        print("Could not extract circle coordinates.
Skipping", key)
                        return
                return [all_points, shape, label]


# get list of json files
with open(json_path.format("json_file_list.txt")) as f:
        json_list = f.read().splitlines()

for i in json_list:
        json_file = json_path.format(i)
        with open(json_file) as f:
                data = json.load(f)
```

```
        regions = {} # dictionary of polygons for this image

        for itr in data:
                filename = data[itr]["filename"]
                print("filename > " + filename)
                num_polys = len(data[itr]["regions"])
                print(f"num polygons > {num_polys}")
                for j in range(num_polys):
                        regions[j] = extract_region(data, itr, j)

        height, width = Image.open(png_path + filename).size
        mask = np.zeros((width, height), dtype = np.uint8)
        for itr in regions:
                try:
                        shape = regions[itr][1]
                except:
                        print("No shape type...")
                        continue
                        #pass
                if shape == "polygon":
                        arr = np.array(regions[itr][0])
                        try:
                                cv2.fillPoly(mask, [arr], colors[regions
[itr][2]])
                        except:
                                print("Unknown label: " + regions[itr]
[2])
                                #cv2.fillPoly(mask, [arr], (255,255,255)
                elif shape == "rect":
                        coords = np.array(regions[itr][0])
                        try:
                                cv2.rectangle(mask,(coords[0], coords
[1]), (coords[2], coords[3]), colors[regions[itr][2]])
                        except:
                                print("Unknown label: " + regions[itr]
[2])
                elif shape == "circle":
                        coords = np.array(regions[itr][0])
                        try:
                                cv2.cirlce(mask, (coords[0], coords
[1]), coords[2], colors[regions[itr][2]])
                        except:
                                print("Unknown label: " + regions[itr]
[2])

        mask[mask == 0] = 255 # change background pixels to default
ignore value
        img = Image.fromarray(mask)
        print(f"saving annotation image > {filename}")
        img.save(output_path + filename, format="png")
```

```
print("\nimage annotations complete")
print("\nflood segments > ", flood_counter)
print("\nflood + cloud segments > ", flood_cloud_counter)
```

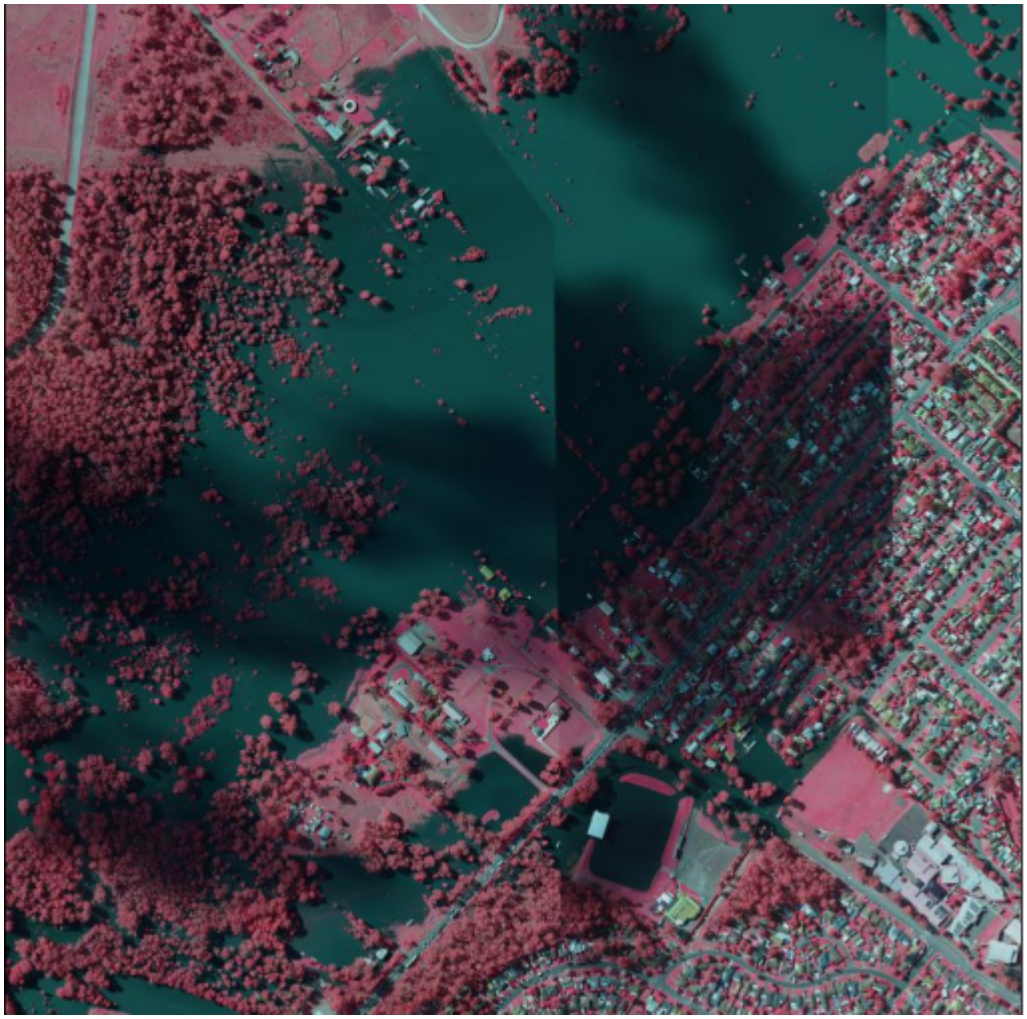
Artificial Inflation of Training Data-set through image augmentation

Initial training jobs of the AWS Semantic Segmentation model did not yield accurate results. This was due to the limited size of the training dataset. To circumvent this problem, the imgaug Python library (https://imgaug.readthedocs.io/en/latest/) was used to artificially inflate the size of the training dataset, thereby improving the accuracy of the output.

This library provides a suite of methods to perform geometric and visual manipulations to the training data-set to create a 'new' images for training. For each geometric transformation applied to an input image, the same transformation is applied to its corresponding annotation image. Non-geometric transformations are not applied to annotation images as they contain data only pertinent to pixel label classifications. Applying a Gaussian blur, for example, to a grayscale annotation image would be semantically meaningless.

A sequence of five augmentations were applied in random order to each square-scaled training image. This was done five times for each image, resulting in a five-fold increase in the training dataset.

Original NRG Image



Original Annotation Image

Below is the Python script used to create the augmented image set.

```
import numpy as np
import imgaug as ia
import imgaug.augmenters as iaa
```

```
from imgaug.augmentables.segmaps import SegmentationMapsOnImage
from PIL import Image

"""
Generate a set of augmented images from the training dataset. Apply
geometric and noise transformations to the images.
For all geometric transformations, apply the same action to all
correspoding annotation images.
"""

train_path = "imgaug_data/train/jpg/"
train_ann_path = "imgaug_data/train_annotation/"

with open(train_path + "train_list01.txt") as f:
        train_list = f.read().splitlines()

with open(train_ann_path + "train_annotation_check_list01.txt") as f:
        train_ann_list = f.read().splitlines()

train_img_list = []

for i in range(len(train_list)):
    #train_ann_img_list = []
    img = Image.open(train_path + train_list[i])
    img = np.array(img)
    train_img_list.append(img)

    img_ann = Image.open(train_ann_path + train_ann_list[i])
    img_ann = np.array(img_ann)
    #train_ann_img_list.append(img_ann)

    seq = iaa.Sequential([
        iaa.Multiply((0.5, 1.5)), # randomly adjust brighness within
range
        iaa.AdditiveGaussianNoise(scale=(0, 0.2*255)), # add Gaussian
Noise
        iaa.GammaContrast((0.5, 0.2)), # adjust contrast of image
        iaa.ScaleX((0.5, 1.5)), # randomly manipulate scale on Y axis
within range
        iaa.ScaleY((0.5, 1.5)) # randomly manipulate scale on X axis
within range
        ], random_order=True)

    segmap = SegmentationMapsOnImage(img_ann, shape=img_ann.shape)

    images_aug = []
    segmaps_aug = []
    for j in range(5):
        images_aug_i, segmaps_aug_i = seq(image=img,
segmentation_maps=segmap)
```

```
        images_aug.append(images_aug_i)
        segmaps_aug.append(segmaps_aug_i)

    for j, image in enumerate(images_aug):
        img = Image.fromarray(image[0])
        img.save(f"imgaug_data/train/jpg/{train_list[i]}_aug_{j}.jpg",
format="JPEG")

    for j, segmap in enumerate(segmaps_aug):
        arr = segmap.get_arr()
        img = Image.fromarray(arr)
        img.save(f"imgaug_data/train_annotation/{train_ann_list[i]}_aug_
{j}.png", format="PNG")

print("Image augmentations complete")
```

AWS SageMaker Semantic Segmentation Results

The notebook for training AWS SageMaker models can be found at 5.1_AWS_Semantic_Segmentation_beta.ipynb. This notebook downloads the Hawkesbury-Nepean and Brewarrina datasets from their S3 bucket storage and divides them into the following groups: `train`, `train_annotation`, `validation`, `validation_annotation`. Multiple training jobs were run to test different algorithmic implementations of semantic segmentation including Pyrmaid-Parsing Scheme, Fully Convolutional Neural Networks, and Deeplab 3.0. As outlined in the Data Preparation page, a supervised machine learning algorithm requires labelled input data, hence why each image has an associated annotation image providing a pixel-level classification. The notebook using this trained model for testing on the Lower Clarence flood data-set can be found at 5.3_AWS_Semantic_Segmentation_pipeline_LC.ipynb.

The below results are the simple output of the Semantic Segmentation Model, a PNG ID mask in which each pixel is given an index value corresponding to their label. The below inference images are the result of combining the flood (index 0) and flood + cloud (index 1) pixels together into a binary truth image.

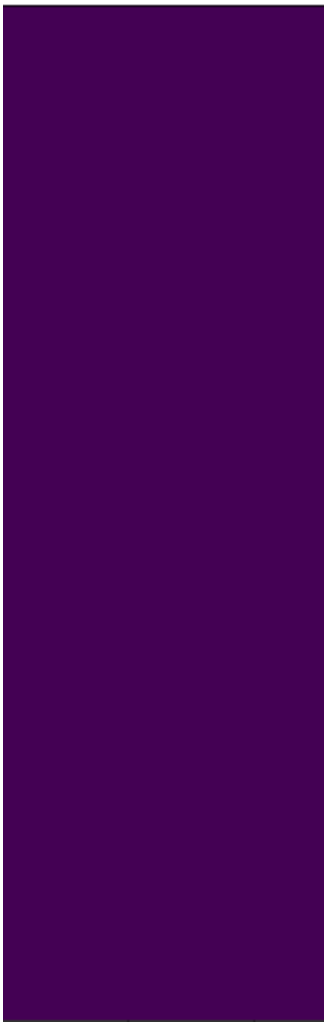
For semantic segmentation tasks the intersection over union (IOU) metric is the appropriate measure for model accuracy. It calculates the amount of overlap between the predicted area and a ground truth image. The ground truth images were generated using a similar approach to the labelled input data with the VGG Image Annotation tool and using a Python script to turn the JSON file into a binary truth image. Team 5 created their own implementation of the IOU metric which can be found in the notebook 4.1_CV_Andrew_NDWI.ipynb.

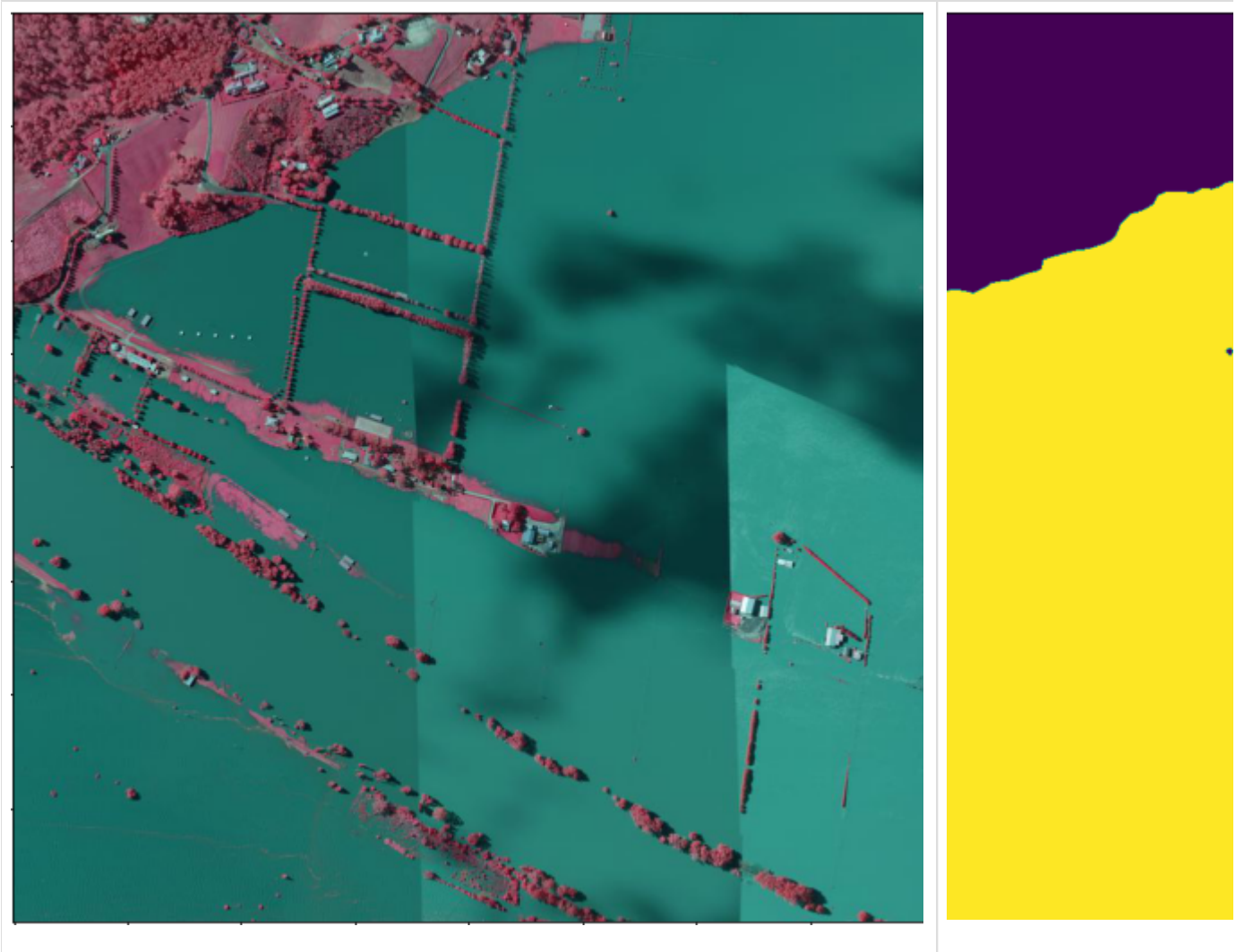
$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

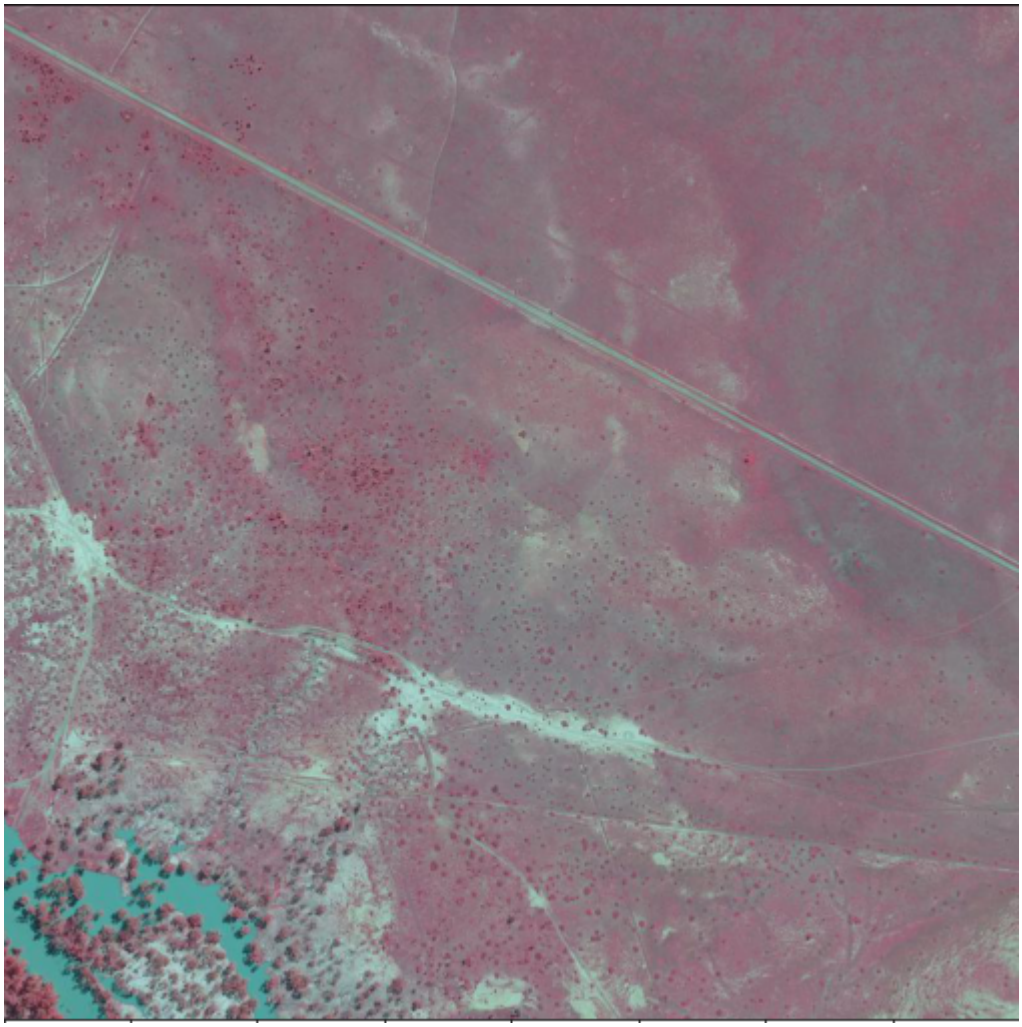
```
def intersection_over_union(binary_img_list, truth_img_list):
    """This method is an implementation of the intersection over union
metric for evaluating the
    accuracy of a predicted image against a truth image. The output
values range from 0 to 1;
    0 representing a complete mismatch, and 1 representing a perfect
match.

    Keyword arguments:
    binary_img_list - a list of binary images generated as predictions
of the target area
    truth_img_list - a list of manually created truth images
    """

    score_list = []

    for i, truth_img in enumerate(truth_img_list):
        binary_img = binary_img_list[i]
        intersection = np.logical_and(binary_img, truth_img)
        union = np.logical_or(binary_img, truth_img)
        intersection = intersection.reshape((-1))
        union = union.reshape((-1))
        total = 0
        count = 0
        for j in range(len(union)):
            if union[j] == True:
                total += 1
                if intersection[j] == True:
                    count += 1
        print("count > " + str(count) + "\t total > " + str(total) +
"\t iou > " + str(count/total))
        score_list.append(str(count/total))

    return score_list
```

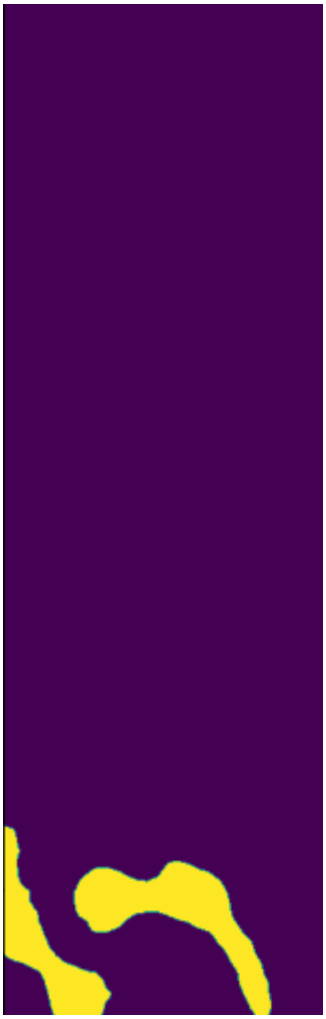
Minimum 15 epochs

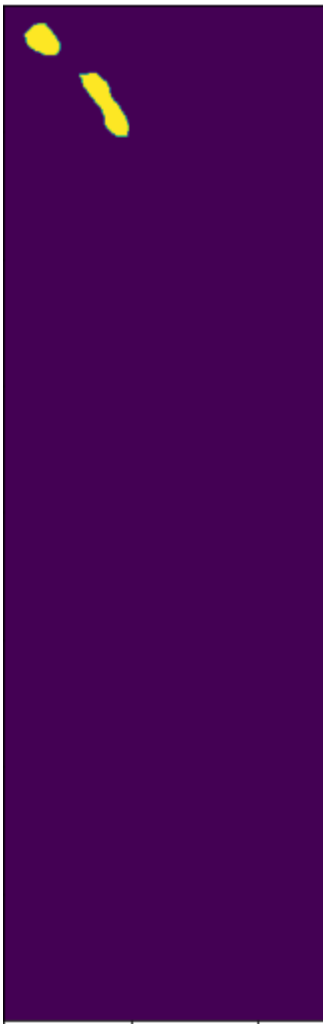
Input Image	Deeplab V3

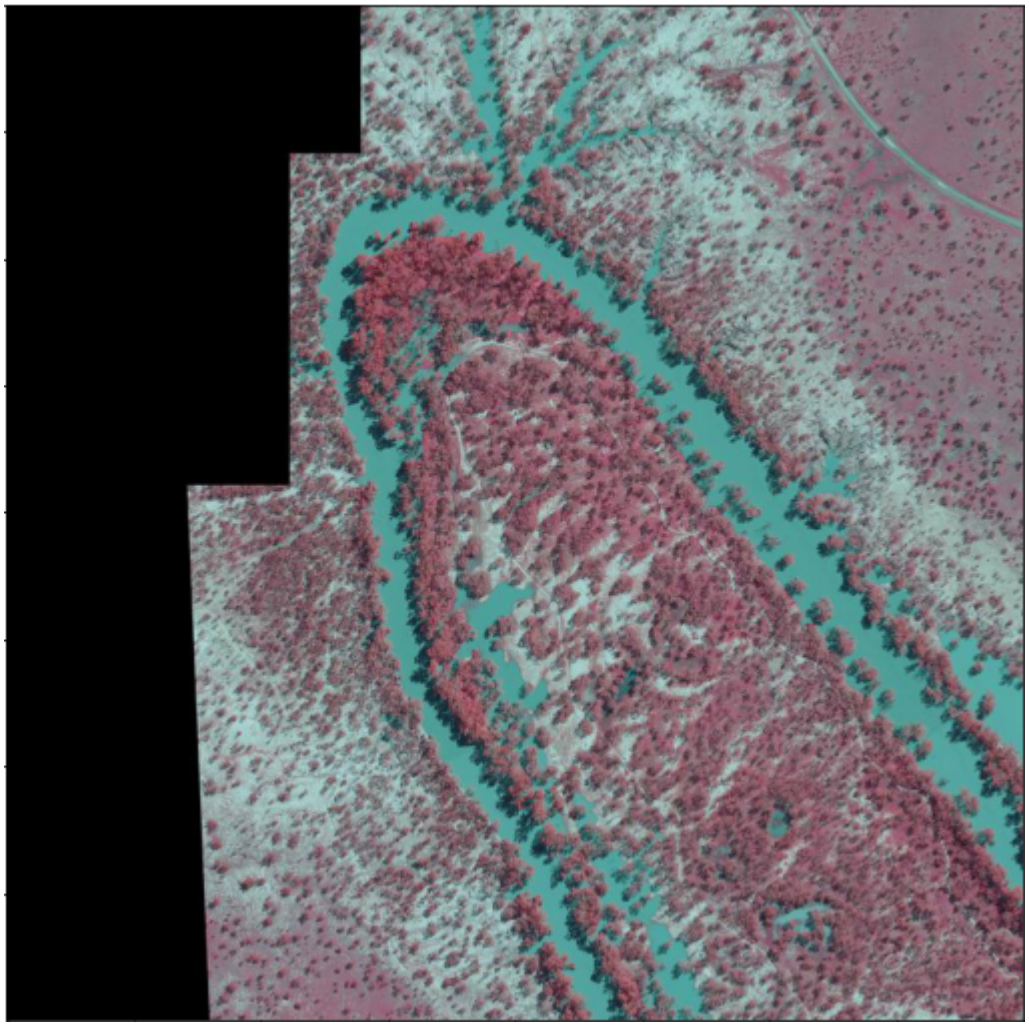






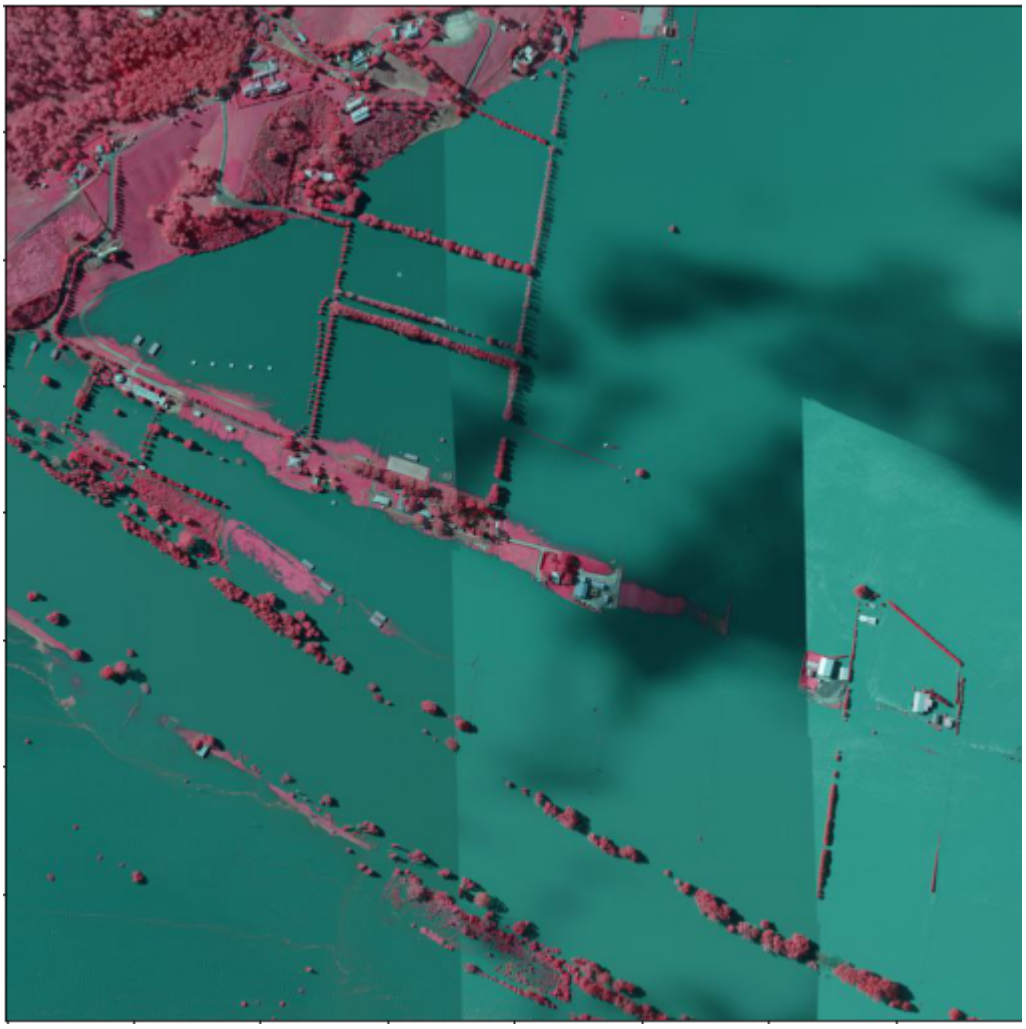


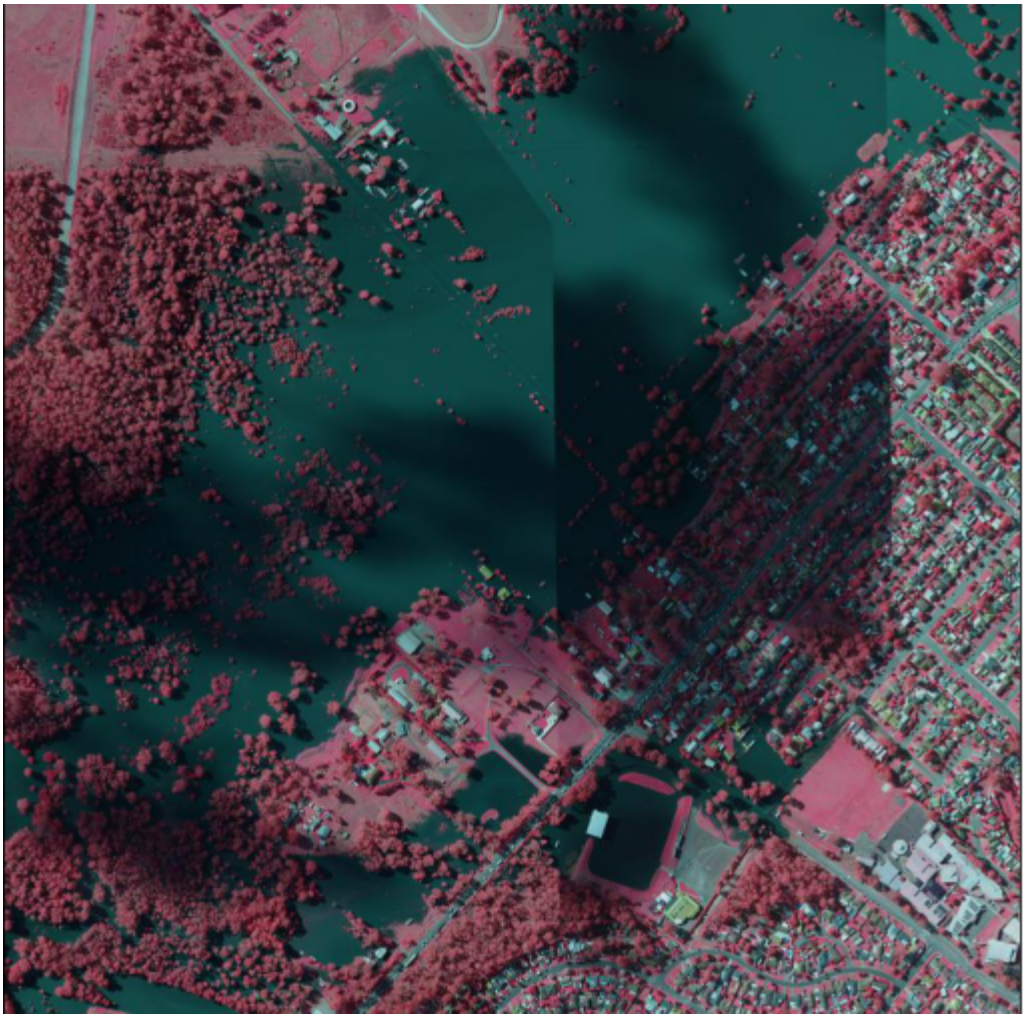


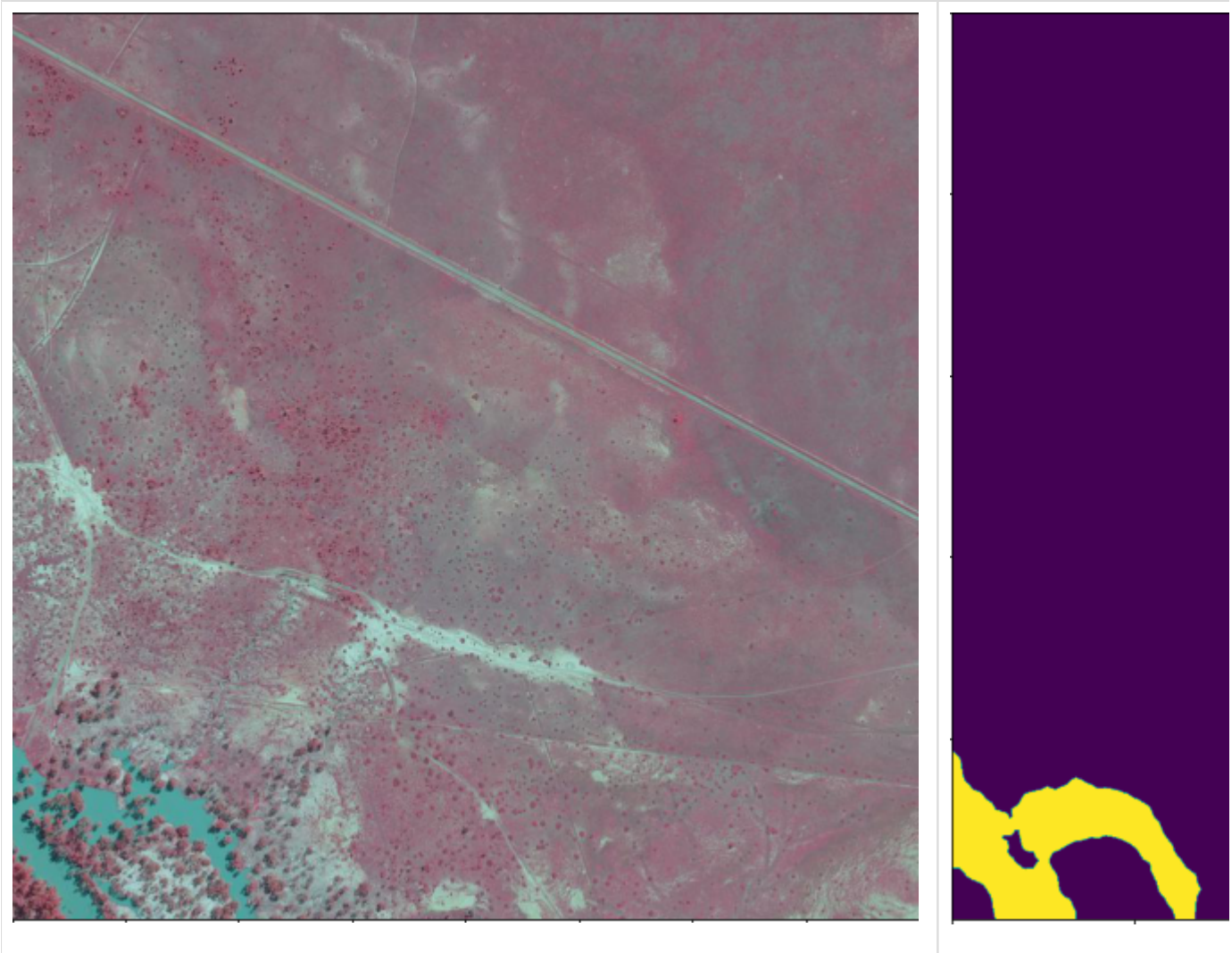


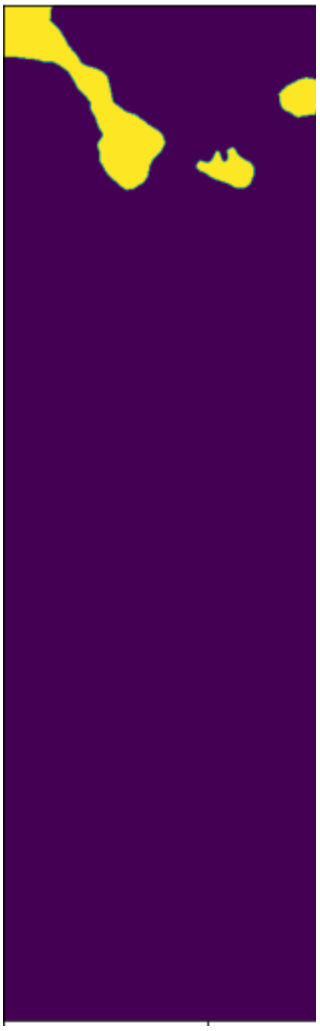
Minimum 10 epochs

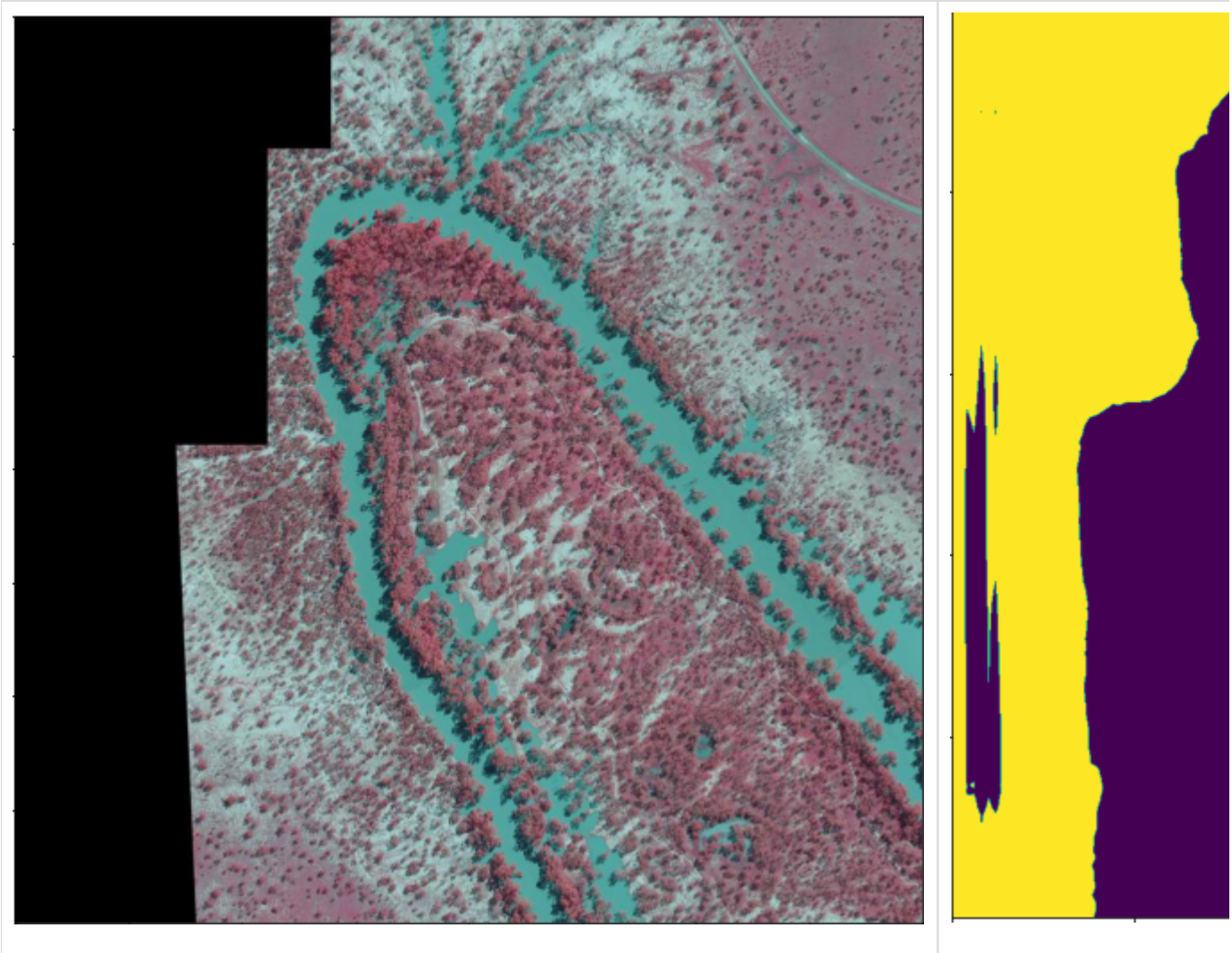
Input Image	FCN











XGBoost & Hybrid Approaches

Introduction

Extreme Gradient Boosted Trees is a learning model used in supervised machine learning. The model in this sense refers to the mathematical structure by which predictions are made from the input. In Introduction to Supervised Machine Learning a linear example was given. In XGBoost decision tree ensembles are used which consist of a set of classification and regression trees (CART) and sums the predictions of a particular output over multiple trees.

XGBoost models are trained in a similar way to the three algorithmic implementations of semantic segmentation provided by AWS SageMaker. A labelled data-set is divided into train and test subsets. An objective function with a loss term and a regularization term is defined and the model parameters that minimize the objective function score are selected. Unlike the linear example given earlier in which the parameters that define some hyperplane in n-1 dimensional space were searched for, the parameters for this learning task are the functions that define and structure of the trees and the leaf scores.

The XGBoost algorithm can be applied to semantic segmentation tasks by training it to classify input pixels in the NRG color space into one of the five labels outlined in Data Preparation. Unlike the AWS Semantic Segmentation pipeline, this approach involves an additional step of extracting features from the pixel channels using the top layers of a convolutional neural network trained on the open VGG ImageNet data-set. This expands the input feature vectors from 3 dimensions, corresponding to the 3 pixel channels, to a 64 dimensional feature vector. These are then used as input into the XGBoost algorithm which classifies these feature vectors into one of the five labels. A tutorial of on this can be found here. A notebook which tests out this approach for flood extraction can be found at 6.4_XGBOOST_VGG(4).ipynb.

Hyperparameter Tuning

There are numerous hyperparameters involved in the training of an XGBoost model. Tuning such parameters in a way that minimises the objective function is a standard step in supervised machine learning tasks. A notebook performing this task can be found at 6.8_XGB_HP_TUNING.ipynb. The parameters tuned are:

- eta - This corresponds to the learning rate and controls how strongly parameters are updated during each iteration of the learning algorithm. An update or ‘step’ that is too large may overshoot some local minima for the objective function, whereas one that is too small may get stuck in a local minima which is not also the global minima.
- gamma - The minimum loss reduction required to add a further partition in a decision tree. Higher values lead to more conservative models.
- lambda - This parameter influences the regularization term, low values are correlated with high training accuracy at the cost of generalization.
- max_depth - This is the maximum depth of the classification and regression trees.

The optimal hyperparameters for the XGBoost algorithm were found using the AWS SageMaker Hyperparameter Tuning Job API. The results of the training job can be found here.

Training the XGB Model

Once the data is split into the train and test datasets and the hyperparameters have been determined, training the model on the data only requires a few lines of code as shown below.

```
xgb_model = xgb.XGBClassifier(nthread=-1,
                           gpu_id=-1,
                           booster='gbtree',
                           learning_rate=eta,
                           colsample_bylevel=1,
                           colsample_bynode=1,
                           colsample_bytree=1,
                           gamma=gamma,
                           reg_alpha=alpha,
                           reg_lambda=_lambda,
                           max_delta_step=max_delta_step,
                           max_depth=max_depth,
                           min_child_weight=min_child_weight,
                           importance_type='gain',
                           interaction_constraints='',
                           missing=np.nan,
                           monotone_constraints='()',
                           n_estimators=20,
                           n_jobs=0,
                           num_parallel_tree=1,
                           objective='multi:softprob',
                           random_state=42,
                           scale_pos_weight=None,
                           subsample=1,
                           tree_method='exact',
                           validate_parameters=1,
                           verbosity=None)

xgb_model.fit(X=X_for_training, y=Y_for_training, eval_set=
[(X_for_test, Y_for_test)], eval_metric='mlogloss',
early_stopping_rounds=5)
```

The output of this model is a multi-dimensional array of size m x n x 5 where m is the number of pixels along the width of the input image and n is the number of pixels along the height of the input image. The 5 values at each m x n data-point represent the probability of that pixel belonging to `flood`, `flood + cloud`, `trees`, `buildings`, or `land` respectively. Obtaining a binary image of the predicted flood area is a simple matter of determining which pixels in the image have the highest probability of belonging to the values at `[m,n,0]` and `[m,n,1]` . This is done in the code below.

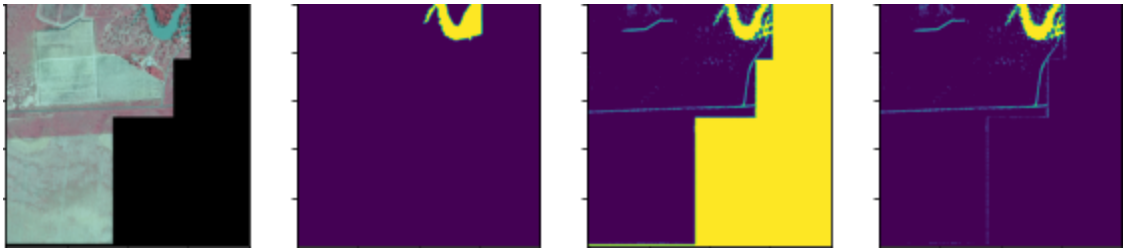
```
xgb_proba = xgb_model.predict_proba(test_features)
xgb_proba = xgb_proba.reshape((xgb_img_list.shape[0], xgb_img_list.shape
[1], xgb_img_list.shape[2], 5))
print(xgb_proba.shape) # (10000,10000,5)

xgb_flood = []

xgb_flood_clusters = (0,1)

for output in xgb_proba:
    hp_joined_img = np.greater(output[:,:,xgb_flood_clusters[0]],0.15)
    hp_joined_img = np.logical_or(hp_joined_img, np.greater(output[:,:,
xgb_flood_clusters[1]],0.15))
    xgb_flood.append(hp_joined_img)
```

Post-processing

For some tiled images from the Brewarrina and Hawkesbury Central datasets there was a pattern of false positives. This occurred on edge images which contained some portion of black space that did not have any pixel channel data. The XGBoost model incorrectly classified this area as belonging to the `flood` label. To account for these false positives some post-processing was necessary.

First, a binary image was generated for each image. This binary image had values of 1 for each pixel which correlated to a grayscale value of 0 in the original NRG images. These negative mask images were then compared with the binary output of the flood images.

```
def remove_noise(comp_img, neg_img):
    for i in range(comp_img.shape[0]):
        for j in range(comp_img.shape[1]):
            if comp_img[i][j] == True:
                if neg_img[i][j] == True:
                    comp_img[i][j] = False
    return comp_img

neg_mask_list = []
for i in range(len(xgb_img_list)):
    temp_img = cv2.cvtColor(xgb_img_list[i].copy(), cv2.COLOR_BGR2GRAY)
    temp_mask = np.zeros(temp_img.shape)
    temp_mask = temp_img == 0
    neg_mask_list.append(temp_mask)

xgb_final = []

for i in range(len(xgb_flood)):
    comp_img = xgb_flood[i].copy()
    neg_img = neg_mask_list[i]
    temp_img = remove_noise(comp_img, neg_img)
    xgb_final.append(temp_img)
```



Output and Combination with GMM

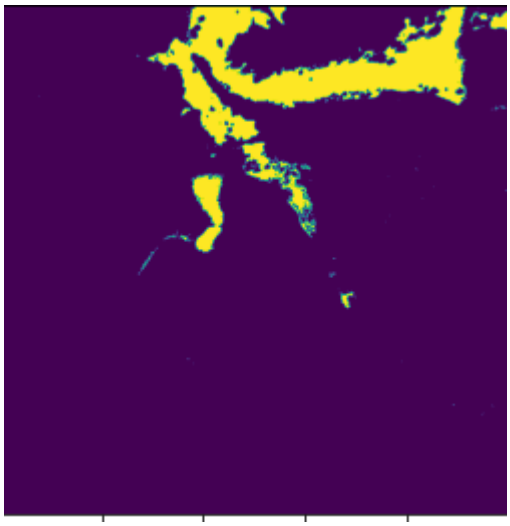
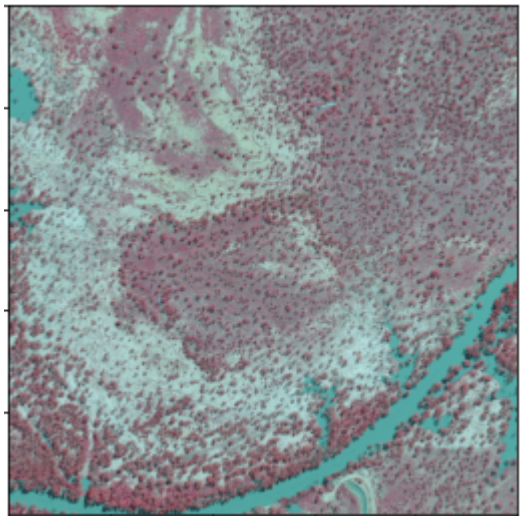
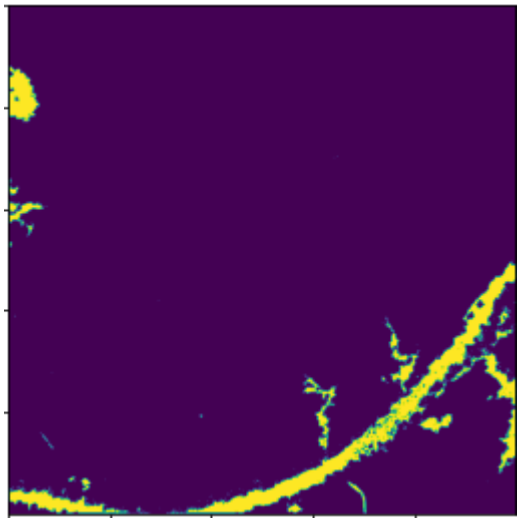
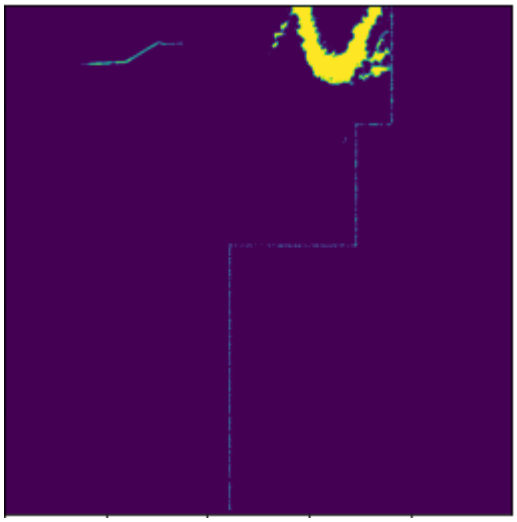
As with the AWS SS pipeline, the appropriate metric to evaluate the output is the IoU. The output of the XGB model was compared with the Gaussian Mixutre Model (GMM) model developed in a separate pipeline. Like the XGB model, the GMM model also returned an array of probabilities summing to 1 for each pixel belonging to one of 13 clusters, 4 of which were manually identified as flood clusters. The GMM is an unsupervised machine learning task, the groupings do not have a semantic meaning as with supervised learning.

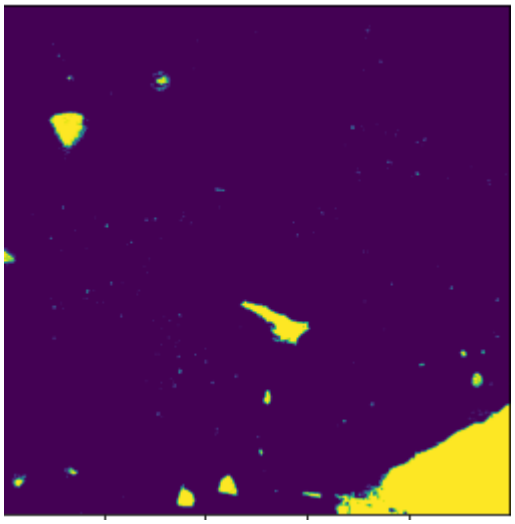
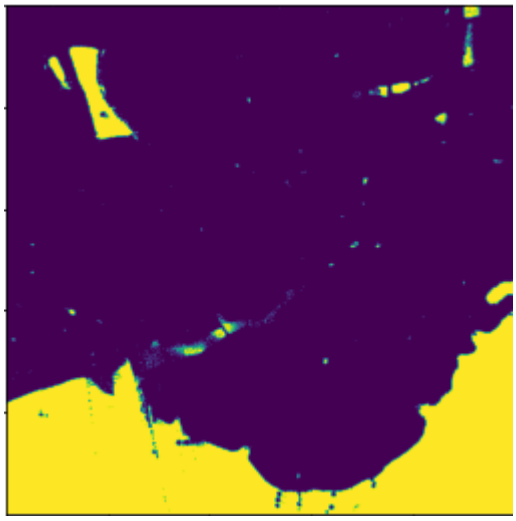
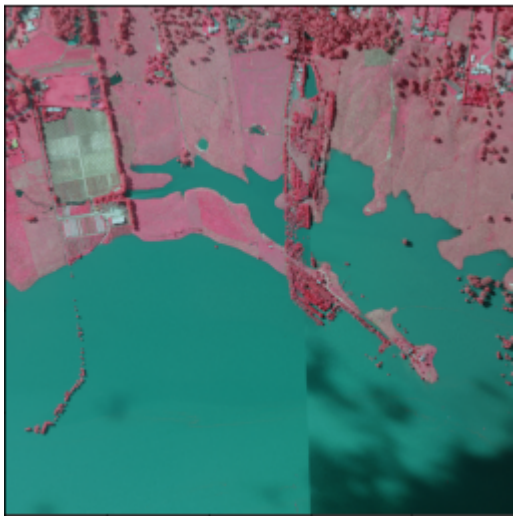
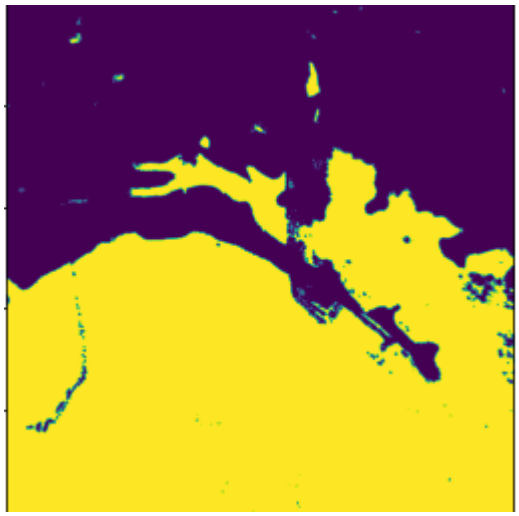
The output of both the GMM and XGB models were combined into a hybrid model. This was a crude averaging of the probabilities of each flood cluster: two from the xgb model (`flood`, `flood + cloud`) and clusters at indices [1,3,4,5] from the GMM model. The code below was used to combine the model together. The notebook for this pipeline can be found at 6.9_XGB_GMM_HYBRID.ipynb.

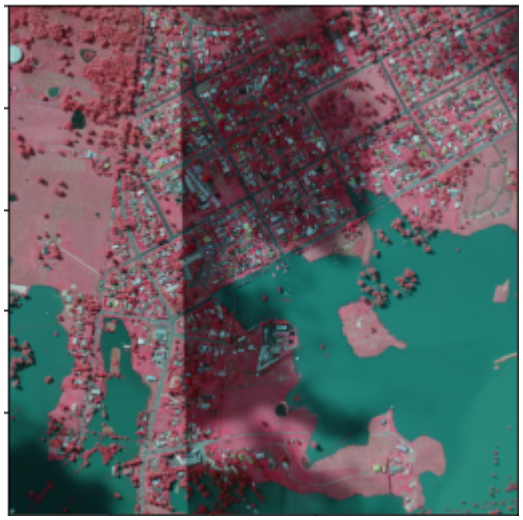
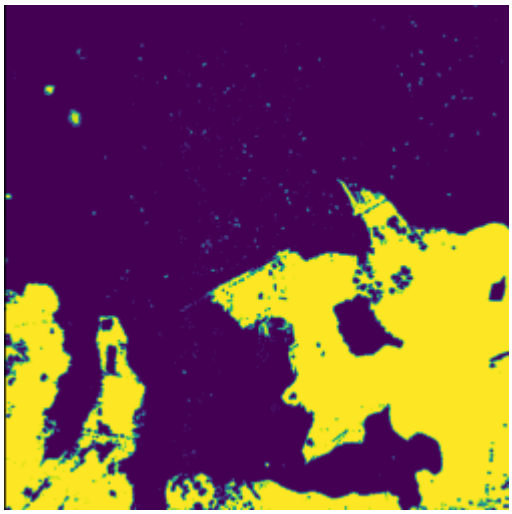
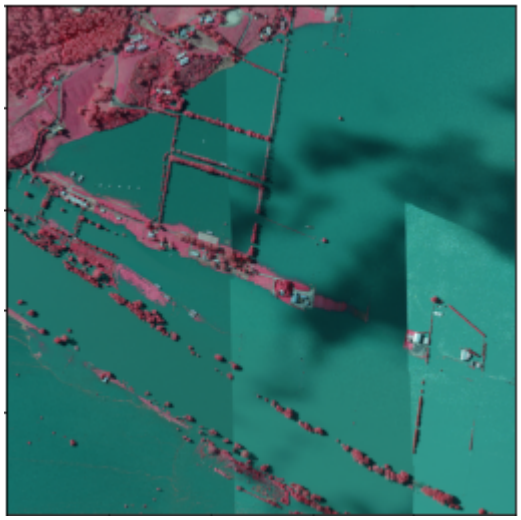
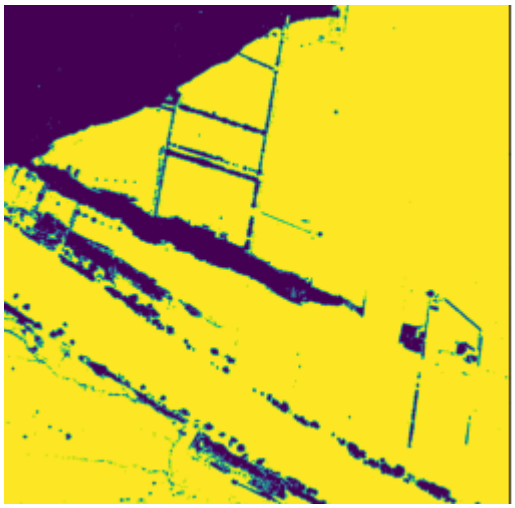
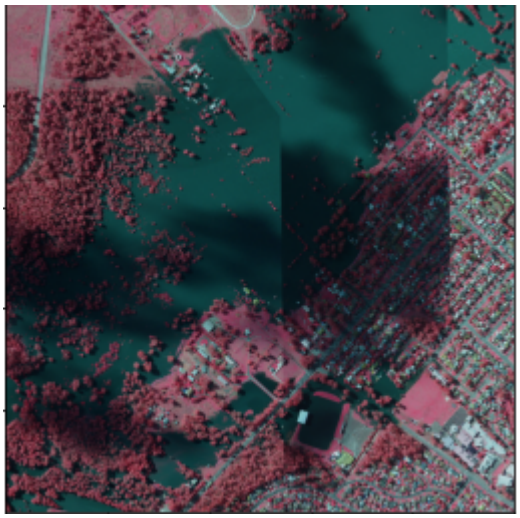
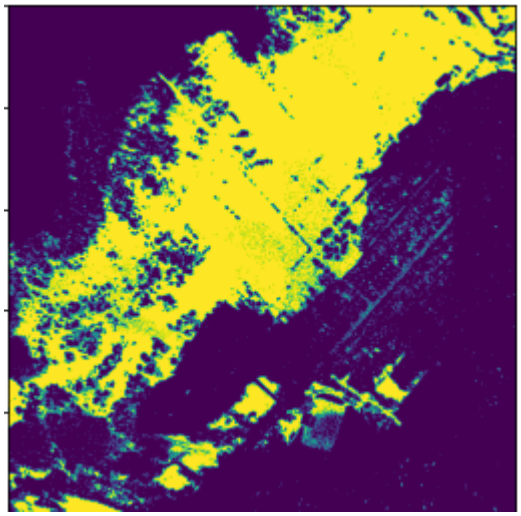
```
avg_img_list = []

for i in range(len(xgb_proba)):
    xgb_proba_01 = xgb_proba[i][:,:,xgb_flood_clusters[0]]
    xgb_proba_02 = xgb_proba[i][:,:,xgb_flood_clusters[1]]
    gmm_proba_01 = gmm_proba_list[i][:,:,gmm_flood_clusters[0]]
    gmm_proba_02 = gmm_proba_list[i][:,:,gmm_flood_clusters[1]]
    gmm_proba_03 = gmm_proba_list[i][:,:,gmm_flood_clusters[2]]
    gmm_proba_04 = gmm_proba_list[i][:,:,gmm_flood_clusters[3]]
    avg_img = (xgb_proba_01 + xgb_proba_02 + gmm_proba_01 +
gmm_proba_02 + gmm_proba_03 + gmm_proba_04) / 6
    avg_img_list.append(avg_img)

avg_img_list = np.array(avg_img_list)
print(avg_img_list.shape)
```

Below is a table of the IoU scores of the XGB, GMM, and Hybrid models on a subset of 9 truth images.

Input Image	XGB IoU	GMM IoU	Hybrid IoU	Hybrid output

	0.694	0.015	0.702	
	0.665	0.032	0.562	
	0.574	0.038	0.585	

	0.464	0.746	0.747	
	0.780	0.887	0.888	
	0.962	0.964	0.965	

	0.774	0.851	0.853	
	0.933	0.913	0.919	
	0.656	0.665	0.667	
Average	0.722	0.568	0.765	

References

Title page cover photo.
Rosalie flooded. (2011). Retrieved from https://www.abc.net.au/news/2011-06-03/brisbane-floods3a-before-and-after-article/2742794?nw=0