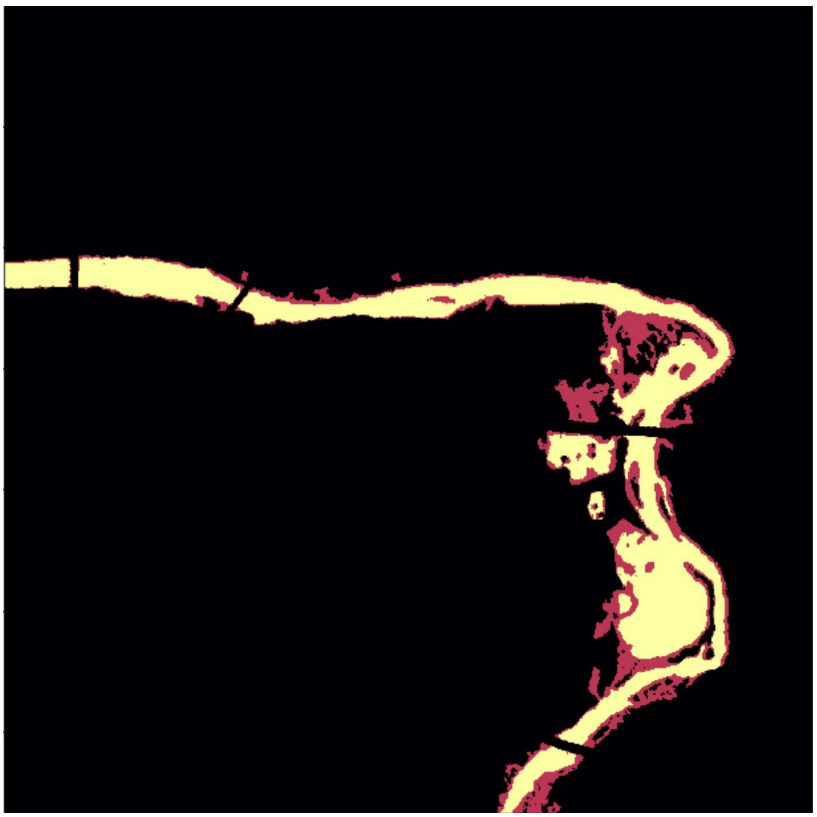
BUREAU OF RECLAMATION

Seasonal/Temporary Wetland/Floodplain Delineation using Remote Sensing and Deep Learning

Science and Technology Program
Research and Development Office
Final Report No. ST-2020-1867-01



U.S. Department of the Interior

March 12, 2021

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.			
1. REPORT DATE *(DD-MM-YYYY)* 12-03-2021	2. REPORT TYPE Research		3. DATES COVERED *(From - To)* 2017-10-01 – 2020-09-30
4. TITLE AND SUBTITLE Seasonal/Temporary Wetland/Floodplain Delineation using Remote Sensing and Deep Learning			5a. CONTRACT NUMBER 20XR0680A1-RY15412018WP31883 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER 1541 (S&T)
6. AUTHOR(S) Vanessa King, Hydrologist			5d. PROJECT NUMBER Final Report No. ST-2020-1867-01 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Planning Division, Decision Analysis Branch California-Great Basin Regional Office Bureau of Reclamation U.S. Department of the Interior 2800 Cottage Way, W-2830 Sacramento, CA 95825			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Science and Technology Program Research and Development Office Bureau of Reclamation U.S. Department of the Interior Denver Federal Center PO Box 25007, Denver, CO 80225-0007			10. SPONSOR/MONITOR'S ACRONYM(S) Reclamation 11. SPONSOR/MONITOR'S REPORT NUMBER(S) Final Report ST-2020-1867-01
12. DISTRIBUTION/AVAILABILITY STATEMENT Final Report may be downloaded from https://www.usbr.gov/research/projects/index.html			
13. SUPPLEMENTARY NOTES			
14. ABSTRACT Seasonal wetlands are an important habitat for many aquatic species, including juvenile anadromous fish. In the past, the delineation of seasonal wetlands has been very limited and often inaccurate by traditional methods used by Reclamation. A new methodology was created to automatically delineate seasonal wetlands from satellite imagery using machine learning methods. While additional research is needed to verify the accuracy of the results, this methodology has the potential to identify seasonal wetlands over a large area at a much lower cost than traditional methods.			
15. SUBJECT TERMS Remote sensing, wetlands, machine learning			
16. SECURITY CLASSIFICATION OF:		17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES 19a. NAME OF RESPONSIBLE PERSON
a. REPORT U	b. ABSTRACT U	THIS PAGE U	19b. TELEPHONE NUMBER *(Include area code)*

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Mission Statements

The Department of the Interior (DOI) conserves and manages the Nation's natural resources and cultural heritage for the benefit and enjoyment of the American people, provides scientific and other information about natural resources and natural hazards to address societal challenges and create opportunities for the American people, and honors the Nation's trust responsibilities or special commitments to American Indians, Alaska Natives, and affiliated island communities to help them prosper.

The mission of the Bureau of Reclamation is to manage, develop, and protect water and related resources in an environmentally and economically sound manner in the interest of the American public.

Disclaimer

Information in this report may not be used for advertising or promotional purposes. The data and findings should not be construed as an endorsement of any product or firm by the Bureau of Reclamation, Department of Interior, or Federal Government. The products evaluated in the report were evaluated for purposes specific to the Bureau of Reclamation mission. Reclamation gives no warranties or guarantees, expressed or implied, for the products evaluated in this report, including merchantability or fitness for a particular purpose.

Acknowledgements

The Science and Technology Program, Bureau of Reclamation, sponsored this research. Planet Labs, Inc. supported this project by providing free access to aerial imagery. The United States Geological Survey contributed the use of a supercomputer.

Seasonal/Temporary Wetland/Floodplain Delineation using Remote Sensing and Deep Learning

Final Report No. ST-2020-1867-01

prepared by

California-Great Basin Regional Office
Vanessa King, Hydrologist

Cover image: A seasonal wetland near Redding, California, delineated using the methodology developed in this project.

Peer Review

Bureau of Reclamation
Research and Development Office
Science and Technology Program

Final Report ST-2020-1867-01

Report Title

VANESSA KING Digitally signed by VANESSA KING Date: 2021.05.04 21:26:08 -07'00'

Prepared by: Vanessa King
Hydrologist, California-Great Basin Region Planning Division

Peer Review: Michael Wright
Hydrologist, California-Great Basin Region Planning Division

Acronyms and Abbreviations

CNN	Convolutional Neural Network
DBSCAN	Density Based Spatial Clustering of Applications with Noise
GAN	Generative Adversarial Network
HDBSCAN	Hierarchical DBSCAN
JULE	Joint Unsupervised Learning
LiDAR	Light Detection and Ranging
NDMI	Normalized Difference Moisture Index
NDVI	Normalized Difference Vegetation Index
NDWI	Normalized Difference Water Index
PCA	Principal Components Analysis
Planet	Planet Labs, Inc.
Reclamation	Bureau of Reclamation
SNE	Stochastic Neighbor Embedding
t-SNE	t-Distributed Stochastic Neighbor Embedding
UAV	Unmanned Aerial Vehicle
UMAP	Uniform Manifold Approximation and Projection
USGS	United States Geological Survey

Contents

Page

Executive Summary

Seasonal wetlands are an important habitat for many aquatic species, including juvenile anadromous fish. In the past, the delineation of seasonal wetlands by traditional methods used by Reclamation has been very limited and often inaccurate. A new methodology was created to automatically delineate seasonal wetlands from satellite imagery using machine learning methods. The primary intended application of this research was to identify potential habitat for juvenile anadromous fish, but the methodology would be applicable to any project where identification of seasonal wetlands is desired.

Wetland delineation was achieved through use of imagery from the Planet Labs, Inc. (Planet) satellite constellation. Indices were calculated from the wavelength data provided by Planet's satellites for a pair of images of the Redding area, one before and one after a major rainfall event. A multidimensional space was created by combining the two images and providing index values for each pixel on each day. State-of-the-art machine learning methods were applied to the multidimensional space, classifying each pixel as a member of different clusters. Some of these clusters were observed to correspond to areas which were flooded in one image but not the other. The total area of these clusters could be used to calculate an estimate of floodplain acreage.

This method relied upon georeferencing based on manual identification of multiple identical points to correct the limitations of Planet's georeferencing. It also relied upon manual identification of useful images to compare flooded against dry states and upon analysis of the clustered results to identify clusters representing flooded pixels. It is possible that other image sources such as Sentinel-2 would have better georeferencing applied to the images before being provided to the public, obviating the need additional processing before applying the classification method. The method tested on the Redding area could be applied to other areas but manual identification of flooded and dry images, analysis of the clusters outputted to find clusters corresponding to flooded areas (which may involve iterative improvement of the clustering algorithm) and potentially manual georeferencing would have to be conducted. Application of the method more generally, for example to a broader set of imagery of a larger area, would require automatization of these manual processes.

While additional research is needed to verify the accuracy of the results, this methodology has the potential to identify seasonal wetlands over a large area at a much lower cost than traditional methods.

1. Introduction

1.1 Author's Note

The Principal Investigator of this project, Zackary Leady, left Reclamation before completing this report. As a result, this report has been compiled in his absence using a limited set of available documentation.

1.2 Background and Research Goals

Anadromous fish such as Chinook salmon are born from eggs laid in gravelly stream beds and achieve maturity while traveling toward the ocean down wider and wider streams. They live their adult lives in the oceans, where they are harvested by commercial fisheries and other predators, and return to inland streams in order to breed, supporting recreational fishing along the way. Populations of anadromous fish rearing in California streams have dropped in the last century as dams, levees, and other human structures have blocked or degraded their habitat. This decrease has harmed economic and environmental interests in California and elsewhere, and much effort continues to be exerted to increase anadromous fish populations.

Among these efforts is habitat restoration downstream of spawning areas, where juveniles feed and grow. If too many juveniles are produced by hatcheries and natural spawning for the available habitat, the nutrition of the fish will suffer and deaths from not only malnutrition but downstream predation and other causes will increase; juvenile habitat availability can become a bottleneck on the entire system. Because availability of floodplain habitat has been correlated with increased growth in juveniles and is recognized as being 'in critically short supply', this often takes the form of increasing available floodplain. Estimation of the area of juvenile habitat available is therefore important for modeling anadromous fish populations or developing habitat restoration plans (Moyle et al., 2008).

It is difficult, however, to accurately assess the available juvenile floodplain habitat at any given place and time. Floodplains are by nature ephemeral, and their extent must be measured in time as well as space. Inundated areas not attached to the main stem, meanwhile, are not valid habitat. Satellite imagery has been used to estimate floodplain extent using Landsat and other satellites, but imagery for any given site is only captured rarely. More recently, a denser network of smaller satellites has been developed by Planet called PlanetScope (Planet Labs, 2021). Imagery from Planet's satellites offer more frequent images of any given location, and have been used in floodplain estimation in other areas of the world (Cooley et al., 2017). Our goal is to estimate juvenile habitat availability along Central Valley streams in recent years while assisting in estimation of floodplain in future seasons using this dataset.

2. Literature Review

A literature review was undertaken to better understand the current knowledge regarding various topics related to the development of a deep learning algorithm, including machine learning, distance metrics, dimensionality reduction, clustering, supervised learning, and unsupervised learning. The results of this literature review are summarized below.

2.1 Machine Learning

Machine learning is an analysis method in which neural networks of varying structures take data as an input and output categorizations or other labelings that represent an algorithmic understanding of the underlying structure of the data. This offers a compelling alternative to manual expert review of thousands of satellite images. In particular, the sub-field known as 'deep learning', because its networks are built out of multiple layers, has seen success in the last decade and has been incorporated into processes used by computing giants including Microsoft, Google and Facebook. In Unsupervised Convolutional Neural Networks (CNNs), the deep, complicated network can take many different shapes but generally repeatedly performs mathematics similar to convolution, resulting in the synthesis of information about a small region and the passing on of this information to a higher-level analytical network. CNNs have shown remarkable aptitude at problems such as image segmentation and identification (e.g. identifying whether a picture contain a cat, a dog, or both, and which pixels are part of the cat object, the dog object, or neither) despite being unsupervised by human intelligence during the computational process. Conversely, analysis of remote sensing imagery such as Planet data has often been undertaken via supervised methods such as hyperspectral image classification, but this requires the existence of a large set of labeled training data (Zhu et al., 2017, Zhang et al., 2016, LeCun et al., 2015).

2.2 Distance Metrics

Many distance measures, the most familiar of which is the Euclidian distance, exist to calculate differences and similarities between individual pixels. For example, the Minkowski distance is a generalization of the Euclidian method where the exponent can be varied, while Mahalanobis distance uses the covariance matrix of the data and is equivalent to squared Euclidian distance for uncorrelated data. It is difficult to determine an absolute ranking from best to worst; different distance measures perform best for different datasets (Kumar et al., 2014). This is likely due to each metric's tendencies to create clusters with different qualities; for example the Manhattan distance method (distance applied along only the streets of a regular city grid with no diagonal paths, in essence) produces 'hyperrectangular' clusters, which may be desirable if one is clustering items that tend to have rectangular shapes, such as city blocks. Another factor is the tradeoff between simplicity of method and complexity of data returned. Partitioning methods are relatively simple but only contain one layer of information about pixel-to-pixel relationships, while hierarchical methods often have many opaque parameters to be set but are capable of discerning multiple levels of similarity between classes of pixels (Xu and Wunsch, 2005).

2.3 Dimensionality Reduction

Reducing the number of dimensions in a dataset while retaining the most important differences and similarities between the data points is a difficult problem that can nevertheless yield significant gains by simplifying the problem space. Principal Component Analysis (PCA) draws directly from linear algebra principles to essentially re-orient the axes underneath the dataset. First, the axis along which the largest amount of variance exists is identified and each data point is rescaled along this axis or component; then an orthogonal axis explaining the next-largest amount of variance is identified and each data point rescaled; and so on as far as one wishes to go. The benefit of PCA is that, by definition, it orders the new axes of the dataset by importance. The user can then select whatever number of components are justified by balancing computational time and loss of data. PCA is often used to reduce data into a plottable number of dimensions (Wold et al., 1987).

PCA is a linear algorithm, which can result in an inadequate representation of the similarity between data points whose relationships include non-linearity. Among the nonlinear dimensionality reduction techniques created to resolve this issue is Stochastic Neighbor Embedding (SNE), which turns Euclidean distances into conditional probabilities that a point would be picked as a neighbor based on a Gaussian probability density. However, this requires careful parameterization of features such as step size, momentum in gradient descent during optimization, and assumptions of Gaussian noise magnitude and rate of reduction. A further development, t-Distributed Stochastic Neighbor Embedding (t-SNE), uses a cost function that is less difficult to parameterize and assumes a Student-t distribution, not a Gaussian distribution, for low-dimensional similarities. A variety of methods for optimization and tuning exist (van der Maaten and Hinton, 2008).

Uniform Manifold Approximation and Projection (UMAP) is a recent development in dimension reduction using advanced concepts in topology to identify and preserve similarities and differences in the data while embedding it into a smaller dimensional space. It is scalable and avoids computational restrictions on number of dimensions, and in testing on selected datasets has been shown to outperform t-SNE. UMAP is intended to be 'a general purpose dimension reduction technique for machine learning' (McInnes and Healy, 2018).

2.4 Clustering

One key aspect of learning relevant to this task is the choice of a clustering algorithm. These methods attempt to partition the dataset into optimal clusters, which offers a promising platform for the classification of pixels in a satellite image as floodplain or not. Clustering can be performed via hierarchical methods, in which nested groups are created, or via partitional (non-hierarchical) methods, as in the K-family of algorithms. For large datasets, density-based algorithms which require data points offer a more computationally efficient solution (Xu and Wunsch, 2005.

The aforementioned partitional clustering methods are known as the K-family because they require the user to define the number of clusters, K, into which the data is to be partitioned. K-family algorithms use Euclidean distance around initially random cluster centers or centroids, then iterate toward a minimum total Euclidean distance between each point and its cluster's centroid. This method is relatively simple but has difficulty dealing with data in which clusters appear of different sizes and densities, and different results can be produced depending on the K value and initial

centroids selected. The K-medoids algorithm also starts with random selection of 'medoids', but in this case it chooses from amongst the data points and attempts to minimize pairwise distance metrics such as Manhattan distance, which can help avoid the bias toward circular- or oval-shaped clusters engendered by use of squared Euclidian distance (Kumar et al., 2014; Xu and Wunsch, 2005).

Many clustering techniques introduced to overcome the limitations of the K-family partitional algorithms. CLARANS builds on the K-medoids method using graph theory principles, examining randomly chosen neighbor nodes until local optima are found and searching randomly for new candidate medoids. This method can achieve higher quality results but still works in quadratic computational time, making it inefficient for large datasets. DBSCAN (Density Based Spatial Clustering of Applications with Noise) takes a density-based approach in an attempt to avoid the aforementioned difficulties of the K-family of algorithms in dealing with varying densities across the problem space. The user specifies a density threshold and a clustering is built based on the density structure of the dataset, with data points in nearby less dense areas grouped into the clusters (Xu and Wunsch, 2005). A hierarchical clustering method built off of DBSCAN, called HDBSCAN, offers the benefits which redound from learning the hierarchical structure of relationships within one's dataset. Through the hierarchical paradigm different thresholds for inclusion of data points can be applied in different areas, allowing for a more nuanced classification of 'noise' data (Campello et al., 2015). Improvements upon HDBSCAN include 'accelerated HDBSCAN*', which eliminates a distance scale parameter from the set of user choices and reduces the computational run time from quadratic to N log N (McInnes and Healy, 2017).

2.5 Supervised Learning

Supervised learning incorporates labeled training data, accelerating the learning process by offering examples that can be applied to the dataset under examination. This is often used in tasks such as image classification (i.e., is this an image of a cat or a tractor?), image segmentation (which parts of this image, labeled pixel by pixel, are road, which are sky, which are tractor, which are cat, etc.), as well as instance segmentation (which parts of the image are cat #1 and which are cat #2, which sky pixels are clouds and which are clear, separate identification of the left and right treads on the tractor; in general, identifying the boundaries of multiple instances of similar objects).

The DeepLab method applies filtering to different layers of the deep convolutional network, among other contributions, in an attempt to achieve superior image segmentation (L.-C. Chen et al., 2018a). This atrous convolution method continues to be pursued in successor models such as DeepLabv3+ (L.-C. Chen et al., 2018b). Developed in the medical field, the U-Net method applies the convolutional neural network to biomedical image processing, where each pixel in the image must be assigned a class. The method uses 'a u-shaped architecture' in which the original image is processed into a data object with fewer pixels but many more channels per pixel, then re-processed into an image of the same pixel-channel dimensions as the original image. This output is a pixel-by-pixel classification in which each pixel's value is defined by operations involving data from across the image (Ronneberger et al., 2015). This method has been extended to probabilistic segmentation, the generation of multiple equally probable segmentation hypotheses (Kohl et al., 2018). The SegNet method uses non-linear upsampling to eliminate a learning step as pooled information from the original image is decoded into feature classifications. It is designed with efficiency in mind, in terms

of memory, computational load, and number of trainable parameters. This method was inspired by unsupervised learning methods but requires supervised learning to establish the classifications by which the image is to be segmented (Badrinarayanan et al., 2016).

2.6 Unsupervised Learning

Due to the lack of reliable labeled wetland data, this analysis pursued an unsupervised approach. Multiple unsupervised learning methods were considered. The 'W-Net' method is so named because its basic architecture is akin to two applications of the U-Net method: Once to encode unsupervised labels and again to apply them. This represents an attempt to apply the benefits of the U-Net method to pixel classification in an unsupervised setting. This is one example of auto-encoding, an aspect of unsupervised machine learning in which representations are algorithmically found so that dimensionality can be reduced and relationships can be identified which may lead to classification and labeling of pixels in the analyzed imagery (Xia and Kulis, 2017).

The Generative Adversarial Network (GAN) offers another take on deep learning. In its original application it was used for image generation, but in later years it has been applied to unsupervised learning and its structure has been adapted, including the coupling of multiple GANs. In the original version of this paradigm, two neural networks, the Generator and the Discriminator, play a minimax game; the Generator creates images from real and fake data, and the Discriminator attempts to identify each image's provenance. Ideally a Nash equilibrium is reached and, in the original application, synthetic images with an appropriate relationship to the data are created (Bashmal et al., 2018). A sub-class of this method known as MARTA GANs have been used for unsupervised satellite image classification; the Generator learns to create synthetic training images similar to the original data and the Discriminator learns features of the true data (Lin et al., 2017). An information-theoretic extension to GAN called InfoGAN has been developed to maximize "the mutual information between a small subset of the latent variables and the observation" (X. Chen et al., 2016).

Joint Unsupervised Learning (JULE) uses CNNs in a 'recurrent process' paradigm, in which merging and clustering operations constitute the forward pass and CNN-based representation learning constitutes the backward pass. This was originally developed for classification of full images into categories (Yang et al., 2016), but the method has been extended to segmentation of medical images (in other words, labeling of each pixel as a member of one of many groups), including accommodating three-dimensional images (Moriya et al., 2018). Further developments along these lines in the image classification area exist, which report superior classifications on test datasets (Tzoreff et al., 2018).

The aforementioned methods and the purposes for using them often overlap; for example, the original application of JULE did not use K-means clustering to initialize clustering, but it specifies that it could, and the JULE image segmentation method utilizes K-means clustering to reduce dimensionality from the trained CNN to the final image. For this reason, it would be misleading to draw hard lines between the different methods. To generalize, however, it may be fair to summarize these methods as a progression of more and more intricate designs, all building on previous methodologies. While classification of individual images is a portion of this project, ultimately the goal is to string together multiple images into a time series and then cluster based on the estimated

presence or absence of water across time as well as space. Many different methods have been applied across a variety of fields. Distance metrics and other generalized clustering concepts can be adapted to this paradigm, and the selection of a metric and a method (e.g. hierarchical or partitioning) continues to be dependent on the dataset to be analyzed (Aghabozorgi et al., 2015).

3. Methodology and Results

3.1 Image selection and pre-processing

Imagery from Planet was selected for use in this project for the following reasons: (1) the imagery has a high resolution of ~4 meters, (2) Planet has a large constellation of satellites (~150 satellites currently in orbit), allowing for short revisit times of ~1-2 days, and (3) at the time the project was initiated, data were available to government researchers at no cost through the Open California program. The Open California program was terminated in 2019, so if additional imagery is required, it will need to be acquired at cost or from another source. Sentinel-2 (collected by the European Space Agency and distributed by the United States Geological Survey (USGS)) was identified as a potential no-cost alternative source for imagery, but it has lower spatial resolution (~10m) and a slightly longer revisit time (2-3 days). Reclamation's primary point of contact at Planet was Joseph Mascaro.

Approximately 8 terabytes of data were downloaded, covering the domain of the Central Valley Improvement Act (CVPIA) over a period of approximately 2.5 years. These data were stored on the USGS Yeti supercomputer, through a partnership with the USGS. If Reclamation wishes to use these data in the future, they will likely need to be transferred to a Reclamation location.

Two areas with known seasonal wetlands were selected for preliminary analysis: one near Redding, CA, and one including a portion of the Cosumnes River. The former contains both Redding's downtown area and potential floodplain habitat in the Turtle Bay area, while the latter is a managed floodplain area which has been studied and mapped by fisheries biologists (Ribeiro et al., 2004). These two areas are referred to as the Redding area and Cosumnes area, respectively, throughout the report.

3.1.1 Comparison of Top of Atmosphere and Surface Radiance Imagery Products

Planet provides three different imagery projects: Basic Scenes (Level 1B), Ortho Scenes (Level 3B), and Ortho Tile products (Level 3A). Additionally, the Ortho Scene product is available as either of Top of Atmosphere Radiance (at sensor) product or a Surface Reflectance image product (Planet Labs, 2021). The Surface reflectance product is processed to top-of-atmosphere reflectance and then atmospherically corrected to bottom of atmosphere reflectance, which ensures consistency across localized atmospheric conditions and minimizes uncertainty in spectral response across time and location. Standard atmospheric models are used along with MODIS water vapor, ozone, and aerosol data (Planet Labs, 2018).

However, the surface reflectance product is only available worldwide since October 2017, and for selected agricultural regions starting in 2016. For our areas of interest, surface reflectance products were identified starting in November 2016 for the Consumes area and August 2016 for the Redding area.

In order to determine whether the difference in consistency between radiance and reflectance imagery is significant, a pair of images were analyzed from the Redding area that were collected before and during the Carr Fire, which burned July 23 to August 30, 2018.

The selected images were from July 2 and August 14, 2018. As no rain was recorded during this time period, the amount of ground surface change is expected to be minimal. No changes in the ground surface were identified in a visual comparison of the two images. The smoke is clearly visible on the image from August 14, and thus it is expected that the concentration of aerosols will be significantly greater on this image. As the surface reflectance product is corrected for the effect of aerosols, it is expected that the surface reflectance products for these two images will be more similar than the top-of-atmosphere radiance images. Figure 1 shows the selected images in true color.

Figure 2 shows the Band 2 (green) image for top-of-atmosphere radiance product (left) and surface reflectance product (right), for July 2 (top) and August 14 (middle), and the differences between the two dates (bottom). Figure 3 shows the same set of images for Band 4 (near infrared). The green and near-infrared bands are the primary bands of interest in this study.

Table 1 shows the correlation coefficients for the top-of-atmosphere radiance and surface reflectance images. Correlation coefficients are scaled from 0 to 1, with higher numbers indicating that the images are more similar to one another. The comparison shows that the surface reflectance images have the same or slightly lower consistency compared to the top-of-atmosphere radiance images.

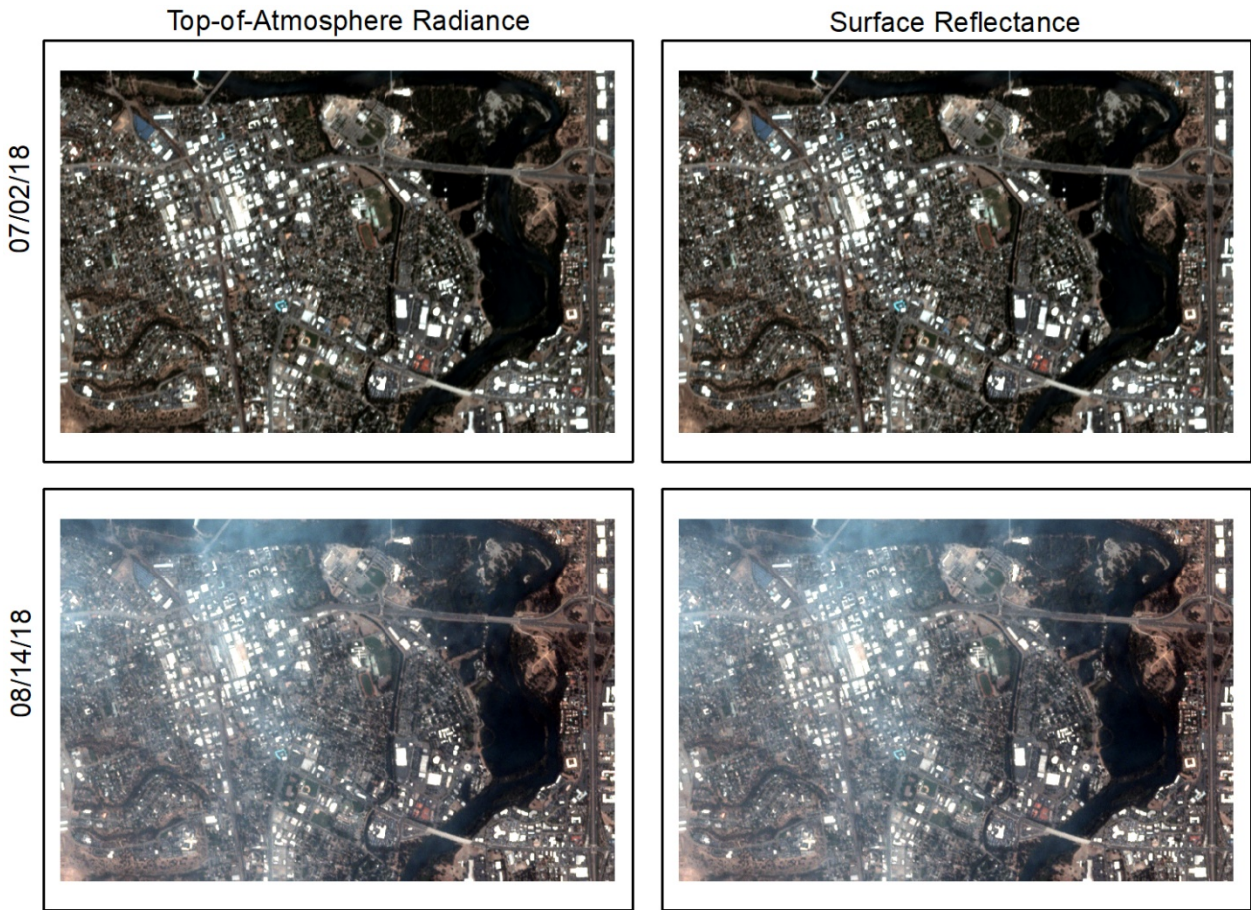


Figure 1: True-color comparison between the top-of-atmosphere radiance and surface reflectance products for the two selected dates (07/02/18 and 08/14/18). Note that for each date, the two products are visually indistinguishable.

Table 1: Correlation coefficients for a comparison between the July 2 and August 14 images, for top-of-atmosphere radiance and surface reflectance products.

Band	Top-of-Atmosphere Radiance	Surface Reflectance
1	0.812	0.808
2	0.865	0.863
3	0.927	0.925
4	0.959	0.959

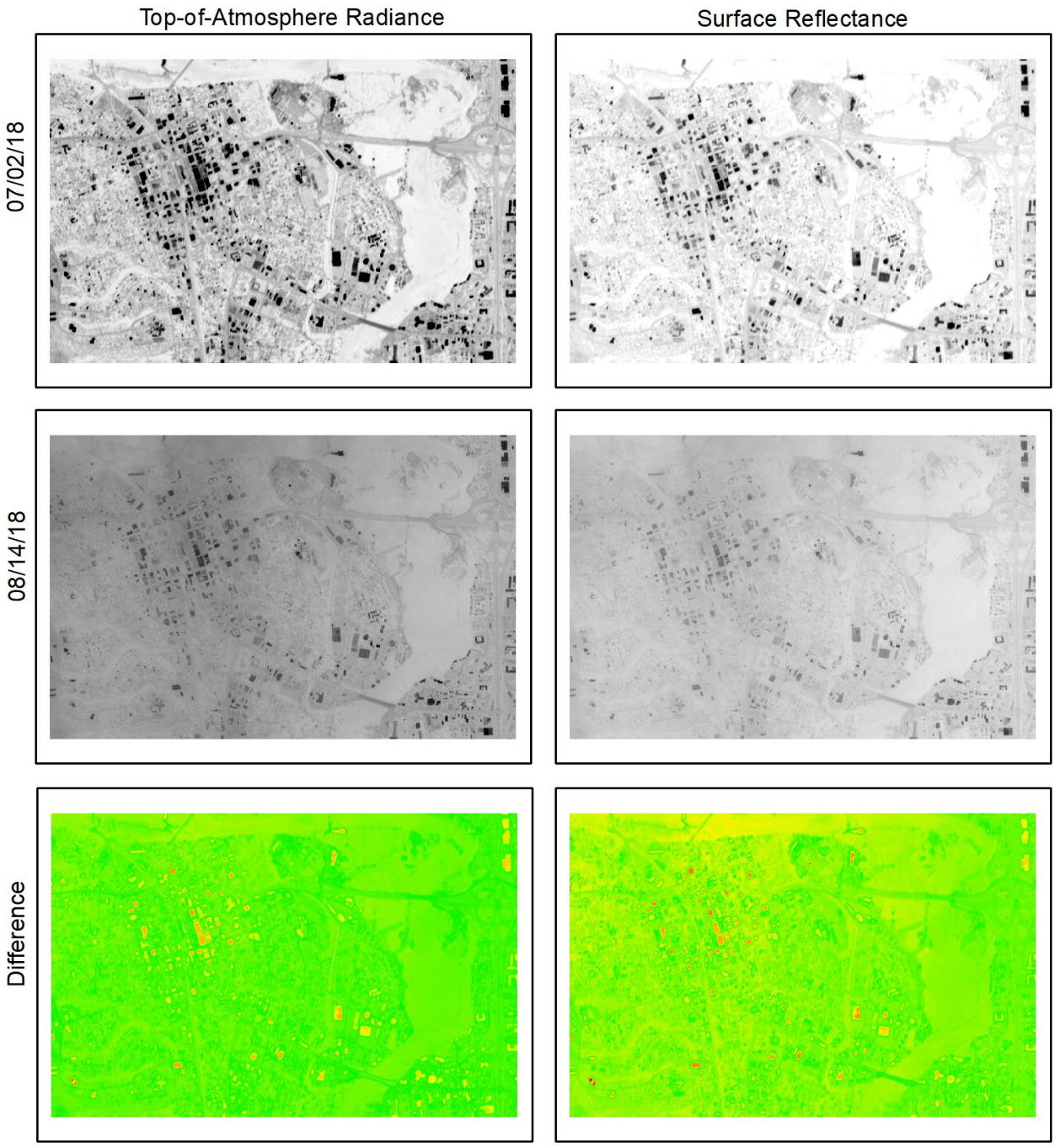


Figure 2: Comparison of Band 2 (green) top-of-atmosphere radiance (left) and surface reflectance (right) products for image taken on July 2, 2018 (top) and August 14, 2018 (middle). The top-of-atmosphere radiance images for each date and surface reflectance for each date are displayed using the same color scale, but the scales for the two sets of data differ as the units are different. The bottom panel shows the absolute value of the difference between the above images. The differences are normalized by the mean value of the 07/02/2018 image products and plotted on the same scale for easier comparison.

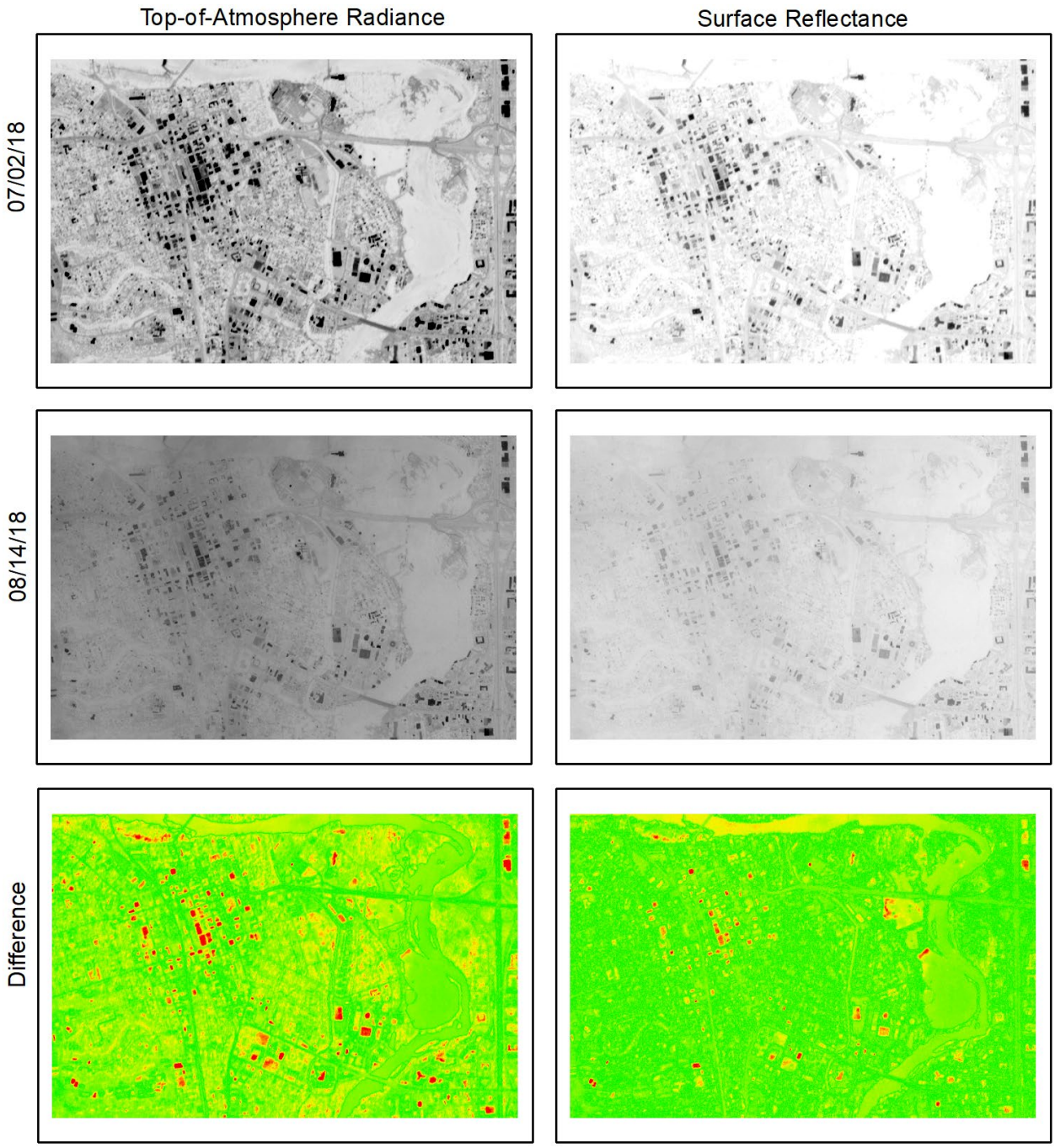


Figure 3: Same as Figure 2, but for Band 4 (near infrared).

Other effects for which the surface reflectance product is corrected include solar zenith angle and water vapor data (Planet Labs, 2018). The vast majority of images for our two study areas were taken within an hour of 12:00 p.m. local time, and for the images taken outside of that time, we were not able to identify any for which a surface reflectance product was provided. Thus, we do not expect there to be a significant effect from the correction for solar zenith angle, especially because the angle is rounded to the nearest 10 degrees for purposes of correction. While the effect of water vapor data may be more significant, it is difficult to separate the effects of water vapor corrections from the ground surface changes that result from precipitation.

While this analysis is limited in scope, it shows that surface reflectance images are no more consistent than top-of-atmosphere radiance images for a pair of images with the same ground surface conditions but different atmospheric images, and thus the choice of which type of images to use is not expected to make a significant difference in our analysis.

3.1.2 Georeferencing

In order to improve the consistency in the geolocations of the Planet images, approximately 50 images were georeferenced to 2016 NAIP imagery, using ArcMap software.

The images were first projected from the original WGS geographic coordinate system to a projected coordinate system, NAD83 UTM Zone 10N, with a pixel size of 4 meters. A 4 km by 4 km area of interest was then selected for the Redding images, and another one for the Cosumnes images. The images were then clipped to these areas of interest, with a 40-meter buffer on each side, in order to ensure that images would still cover the entire area of interest after georeferencing. The areas of interest were selected by calculating the area over which all the images for that location overlapped, and then extending that area to cover more of the areas known to be subject to seasonal inundation, and to reach the desired size. Images that did not cover the known seasonal inundation areas were discarded. The Cosumnes area of interest fell on the border of two coverage areas, so adjacent images were mosaicked prior to georeferencing as necessary.

Twenty-five control points were then selected for each set of images (Redding and Cosumnes). These control points were selected to be adequately distributed throughout the image and to be located at points that were expected to experience minimal change over time and to be identifiable on all images, such as corners of buildings or intersections of roads. Prior to georeferencing, the majority of location differences between the Planet data and the NAIP imagery were within the nominal location accuracy of 10 meters, but some were greater, with one image differing from the NAIP imagery by about 70 meters. Not all control points were used for each image, due to images not covering the full area of interest, or to the control points not being identifiable because of cloud cover or other reasons. Georeferencing was performed using a first-order polynomial (affine) transformation.

For the Redding images, the number of control points used for each image ranged from 15-25, with a mean of 23.6. The root-mean-square (RMS) error ranged from 1.70 m to 4.21 m, with a mean of 2.33 m, with only one image having a RMS error greater than the pixel size of 4 m.

For the Cosumnes images, 18 images used all 25 control points, while the other two used 24 control points, for a mean of 24.9. The RMS error ranged from 2.08 m to 4.25 m, with a mean of 2.95 m. Two images had a RMS error greater than 4 m.

The georeferencing control points for each site are shown in Figure 4 and Figure 5.

Figure 4. Redding georeferencing control points, along with an example image.

Figure 5. Cosumnes georeferencing control points, along with an example image.

3.1.3 Spectral Consistency

As the wetlands analysis is dependent on detecting spectral changes between images due to flooding, it is important that the signal caused by flooding is much stronger than the differences between images due to other causes. Two pairs of test images that were recorded by different satellites on the same day, one pair in the wet season and one in the dry season, were analyzed to quantify the cross-image differences.

Figure 6 show the Cosumnes images for Band 2 (green; left) and Band 4 (near infrared; right). These are the primary bands of interest. The pair of test images are shown in the top and middle, with the bottom image showing the absolute value of the difference between them plotted on the same scale. The two images have correlation coefficients of 0.991 for band 2 and 0.993 for band 4, indicating a high degree of consistency between the images.

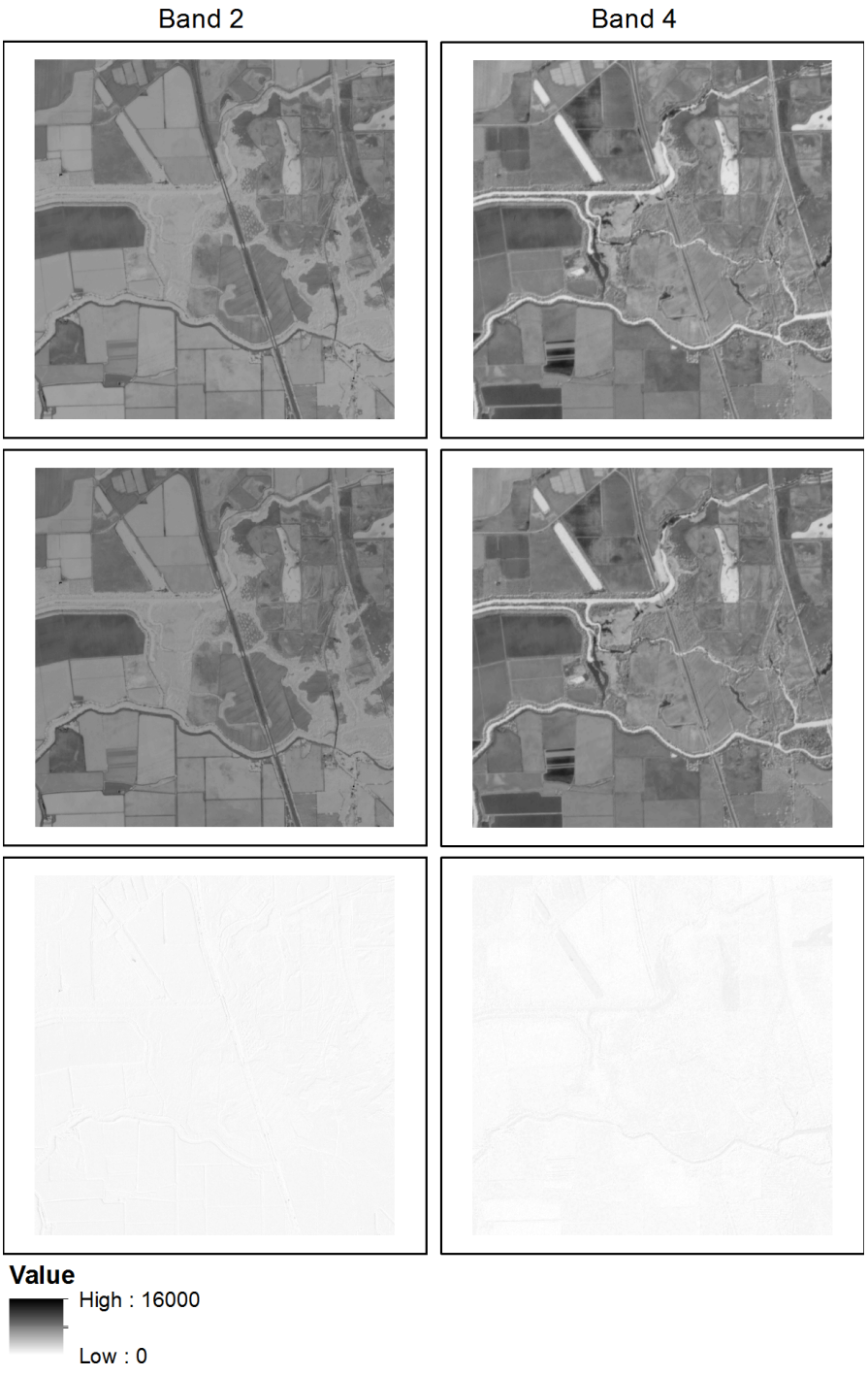


Figure 6: Comparison between Cosumnes images in a dry period. Both images were recorded on 31 August 2016. The top images show the green band (left) and near infrared band (right) for an image recorded by satellite 0e26. The middle images show the green band (left) and near infrared band (right) for an image recorded by satellite 0e03. The bottom images show the absolute value of the difference between the green bands (left) and near infrared bands (right). All images are plotted using the same color scale (bottom).

3.2 Image classification

In order to simplify the problem space, four-band data from Planet's satellites (near infrared, red, green, blue) are synthesized into two indices well established in the remote sensing analysis community. The Normalized Difference Water Index (NDWI) is calculated by dividing green minus near infrared values by the sum of green and near infrared, while the Normalized Difference Vegetation Index (NDVI) is calculated by dividing the difference between near infrared and red values by the sum of those two values. In short, NDWI = (G – NIR) / (G + NIR) and NDVI = (NIR – R) / (NIR + R), where G, R, NIR, stand for green, red, and near infrared, respectively. These indices are not perfect detectors of water and vegetation, but they provide a good start at identifying water (Hussain et al., 2013). and have been applied to Planet data for that purpose (Tetteh and Schönert, 2015; Cooley et al., 2017).

Mapping analyses of wetland data have used data sources including LiDAR (Vondrasek, 2015; Serran and Creed, 2015) Landsat (Quinn and Epshtein, 2013; Quinn and Burns, 2015; Jones, 2015; Pekel et al., 2016; Allen, 2015; Verpoorter et al., 2012; Pasquarella et al., 2016), Worldview (Lane et al., 2014), Synthetic Aperture Radar (SAR) satellites (Martinez and Le Toan, 2007; Schmitt et al., 2018), and of course Planet satellite data (Cooley et al., 2017), as well as combinations of Landsat and Planet data (Gabrielsen, et al., 2016), Landsat and Unmanned Aerial Vehicle (UAV) (O'Brien, 2016), and multiple satellite datasets (Prigent et al., 2016). In general, water masks are formed through analytical processes, often utilizing indices as described above; methods applied to the data range from manual determination of index threshold values to deep learning approaches. These studies offer guidance for our approach.

The JULE method explained above is implemented in Python. This code offers high-level implementations of the complicated processes necessary in creating and using CNNs. A high-quality dataset used in image analysis competitions exists depicting the German city of Potsdam. This includes imagery of the River Havel as well as urban areas, offering a platform for assessment of the method. After confirming a Python workflow that achieved segmentation and labeling of the Potsdam image, the method was applied to the Central Valley imagery that had been obtained from Planet.

The insights gained from working with and reading about the models and concepts above led to the creation of a testbed successfully labeling ephemeral floodplain. The extreme wet season of 2016-2017 offered ample opportunity for identifying images of these areas before and during large flood events. After combining various analytical pathways from among those described above, a method of dimensionality reduction using PCA followed by K-means clustering across pixels defined by the difference between their NDVI and NDWI index values and the average across the images at both time steps resulted in actionable outputs. This method successfully identified many pixels which were dry in one picture and inundated in the next, despite changes in water color (after a major flood event the water is sediment-laden and brown, while in pre-flood conditions it is bluer), potential confounding effects due to some urban surfaces' similar index values to bodies of water, and the possibility of other differences in calibration between the satellite sensors which recorded the images.

Unfortunately, this code contains very limited documentation, but the code is included as Appendix A, to allow for its future use or extension.

4. Future Work

As described above, determination of a suitable distance metric must largely be guided by performance on the target dataset. In order to upscale this effort to encompass larger portions of the Sacramento Valley's potential juvenile salmonid habitat or other areas of interest, the performance of various distance metrics should be evaluated to determine which distance metric or metrics are appropriate for the dataset. The UMAP method described above has promise as a method of reducing dimensionality while maintaining as many of the features of the full dataset as possible. As time series become longer and the number of images under analysis increase, this capacity may be essential to reduce computational complexity.

Of the clustering methods previously mentioned, two partitioning and one hierarchical method are particularly good candidates for future use. K-medoids and K-means are variants on the same relatively simple idea of partitioning, while HDBSCAN is a more complex density-based hierarchical method which may be required if the underlying data structure becomes too complex for the relatively simple computations applied by the K-family of algorithms. If analysis progresses, various clustering methods should be applied to the data to observe which methods best identify spatiotemporal floodplain extent.

Dimensionality reduction can be used not only to reduce computational complexity but also to create a meaningful image in two or three dimensions for visualization by end-users and programmers. UMAP offers this capacity. It is possible that representations of complex multi-image time series classifications generated by UMAP would contain information about the relationship between one type of pixel and another, which in turn could aid the viewer in interpreting a classified image with labeled pixels. This could assist in describing or deciphering the differences and similarities between different types of floodplain, or pixels that are incorrectly classed as floodplain.

While Planet data offer imagery at high spatial and temporal resolution by comparison with other satellite imagery datasets, data quality issues must be considered. Each image is not as well geolocated as the smaller number of images generated by other data sources such as Landsat, meaning that further processing is required before the combination of multiple images into a time series (Cooley et al., 2017). Currently we have utilized manual geolocation corrections on several representative sets of images to correct this issue. This method is scalable to a limited degree; seasons and locations selected by subject matter experts could be addressed with limited turnaround time. Automated methods of addressing this issue would be necessary to scale-up this application.

References

Aghabozorgi, S., Shirkhorshidi, A.S., and Wah, T.Y., 2015. Time-series clustering – A decade review. *Information Systems* 53, 16-38.

Allen, Y., 2015. Landscape Scale Assessment of Floodplain Inundation Frequency Using Landsat Imagery. *River Research and Applications* 32(7), 1609-1620.

Badrinarayanan, V., Kendall, A., and Cipolla, R., 2016. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39(12), 2481-2495.

Bashmal, L., Bazi, Y., AlHichri, H., AlRahhal, M.M., Ammour, N., and Alajlan, N., 2018. Siamese-GAN: Learning Invariant Representations for Aerial Vehicle Image Categorization. *Remote Sensing* 10, 351, 19 pp.

Campello, R.J.G.B., Moulavi, D., Zimek, A., and Sander, J., 2015. Hierarchical density estimates for data clustering, visualization, and outlier detection. *ACM Transactions on Knowledge Discovery from Data* 10(1), 5:1-5:51.

Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K., and Yuille, A.L., 2018a. DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40(4), 834-848.

Chen, L.-C., Zhu, Y., Papandreou, G., Schroff, F., and Adam, H., 2018b. Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation. *ArXiv e-prints.* arXIV:1802.02611.

Chen, X., Duan, Y., Houthoft., R., Schulman, J., Sutskever, I., and Abbeel, P., 2016. InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets. *ArXiv e-prints,* arXiv:1606.03657.

Cooley, S.W., Smith, L.C., Stepan, L., and Mascaro, J., 2017. Tracking Dynamic Northern Surface Water Changes with High-Frequency CubeSat Imagery. *Remote Sensing* 9, 1306, 21 pp.

Gabrielsen, C.G., Murphy, M.A., and Evans, J.S., 2016. Using a multiscale, probabilistic approach to identify spatial-temporal wetland gradients. Remote Sensing of Environment 184, 522–538.

Jones, J.W., 2015. Efficient Wetland Surface Water Detection and Monitoring via Landsat: Comparison with in situ Data from the Everglades Depth Estimation Network. *Remote Sensing* 7, 12503-12538.

Kohl, S.A.A., Romera-Paredes, B., Meyer, C., De Fauw, J., Ledsam, J.R., Maier-Hein, K.H., Eslami, A.S.M., Rezende, D.J., and Ronneberger, O., 2018. A Probabilistic U-Net for Segmentation of Ambiguous Images. *ArXiv e-prints*, arXiv:1806.05034v1.

Hussain, M., Chen, D., Cheng, A., Wei, H., and Stanley, D., 2013. Change detection from remotely sensed images: From pixel-based to object-based approaches. *ISPRS Journal of Photogrammetry and Remote Sensing* 80, 91-106.

Lane, C.R., Liu, H., Autrey, B.C., Anenkhonov, O.A., Chepinoga, V.V., and Wu, Q., 2014. *Remote Sensing* 6, 12187-12216.

LeCun, Y., Bengio, Y., and Hinton, G., 2015. Deep learning. *Nature* 521, 436-444.

Lin, D., Fu, K., Wang, Y., Xu, Guangluan, X., and Sun, X., 2017. MARTA GANs: Unsupervised Representation Learning for Remote Sensing Image Classification. *ArXiv e-prints*, arXiv:1612.08879v3.

Kumar, V., Chhabra, J.K., and Kumar, D., 2014. Performance Evaluation of Distance Metrics in the

Clustering Algorithms. *Infocomp* 13(1), 38-51.
Martinez, J.-M. and Le Toan, T., 2006. Mapping of flood dynamics and spatial distribution of vegetation in the Amazon floodplain using multitemporal SAR data. *Remote Sensing of Environment* 103(3), 209-223.
McInnes, L., & Healy, J., 2017. Accelerated Hierarchical Density Clustering. *2017 IEEE ICDMW*, 33-42.
McInnes, L., & Healy, J., 2018. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *ArXiv e-prints*. arXiv:1802.03426v2.
Moriya, Y., Roth, H.R., Nakamura, S., Oda, H., Nagara, K., Oda, M., and Mori, K., 2018. Unsupervised Segmentation of 3D Medical Images Based on Clustering and Deep Representation Learning. *ArXiv e-prints*. arXIV:1804.03830v1.
Moyle, P.B., Israel, J.A., and Purdy, S.A., 2008. Salmon, Steelhead, and Trout in California: Status of an Emblematic Fauna. Center for Watershed Sciences, University of California, Davis, 316 pp.
O'Brien, T., 2016. Small Unmanned Aerial Vehicles as Remote Sensors: An Effective Data Gathering Tool for Wetland Mapping. Master of Landscape Architecture Thesis, The University of Guelph.
Pasquarella, V.J., Holden, C.E., Kaufman, L., and Woodcock, C.E., 2016. From imagery to ecology: leveraging time series of all available Landsat observations to map and monitor ecosystem state and dynamics. Remote Sensing in Ecology and Conservation, 152-170.
Pekel, J.-F., Cottam, A., Gorelick, N., and Belward, A.S., 2016. High-resolution mapping of global surface water and its long-term changes. *Nature* 540, 418-422.
Planet Labs, Inc., 2018. Planet Surface Reflectance Product. <https://assets.planet.com/marketing/PDF/Planet_Surface_Reflectance_Technical_White_Paper.pdf>. Accessed 10/18/2018.
Planet Labs, Inc., 2021. Planet Imagery Product Specifications. <Planet_Combined_Imagery_Product_Specs_letter_screen.pdf>. Accessed 02/23/2021.
Prigent, C., Lettenmaier, D.P., Aires, F., and Papa, F., 2016. Toward a High-Resolution Monitoring of Continental Surface Water Extent and Dynamics, at Global Scale: from GIEMS (Global Inundation Extent from Multi-Satellites) to SWOT (Surface Water Ocean Topography). *Surv Geophys.* 37, 339-355.
Quinn, N.W.T. and Burns, J.R., 2015. Use of a hybrid optical remote sensing classification technique for seasonal wetland habitat degradation assessment resulting from adoption of real-time salinity management practices. *Journal of Applied Remote Sensing* 9, 096071-2 – 096071-25.
Quinn, N.W.T. and Epshtein, O., 2013. Seasonally-managed wetland footprint delineation using Landsat ETM+ satellite imagery. Environmental Modelling & Software 52, 9-23.
Ribeiro, F., Crain, P.K., and Moyle, P.B., 2004. Variation in condition factor and growth in young-of-year fishes in floodplain and riverine habitats of the Cosumnes River, California. *Hydrobiologia* 527, 77-84.
Ronneberger, O., Fischer, P., and Brox, T., 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. *MICCAI 2015*, 234-241.
Schmitt, M., Hughes, L.H., and Zhu, X.X., 2018. The SEN1-2 Dataset for Deep Learning in SAR-Optical Data Fusion. ISPRS Technical Commission 1 Symposium, 6 pp.
Serran, J.N. and Creed, I.F., 2015. New mapping techniques to estimate the preferential loss of small wetlands on prairie landscapes. *Hydrological Processes* 30(3), 396-409.
Tetteh, G.O. and Schönert, M., 2015. Automatic Generation of Water Masks from RapidEye Images. Journal of Geoscience and Environment Protection 3, 17-23.
Tzoreff, E., Kogan, O., and Choukroun, Y., 2018. Deep Discrimanative Latent Space for Clustering.

NIPS 2017, 9 pp.

Wold, S., Esbensen, K., and Geladi, P., 1987. Principal Component Analysis. *Chemometrics and Intelligent Laboratory Systems* 2, 37-52

van der Maaten, L. and Hinton, G., 2008. Visualizing Data Using t-SNE. *Journal of Machine Learning Research* 9, 2579-2605.

Verpoorter, C., Kutser, T., and Tranvik, L., 2012. Automated mapping of water bodies using Landsat multispectral data. *Limnology and Oceanography: Methods* 10, 1037-1050.

Vondrasek, C., 2015. Delineating forested river habitats and riparian floodplain hydrology with LiDAR. M.S. Thesis, University of Washington.

Xia, X. and Kulis, B., 2017. W-Net: A Deep Model for Fully Unsupervised Image Segmentation. *ArXiv e-prints*, arXiv:1711.08506v1

Xu, R., and Wunsch II, D., 2005. Survey of Clustering Algorithms. *IEEE Transactions on Neural Networks* 16(3), 645-678.

Yang, J., Parikh, D., and Batra, D., 2016. Joint Unsupervised Learning of Deep Representations and Image Clusters. *ArXiv e-prints.* arXiv:1604.03628v3

Zhang, L., Zhang, L., and Du, B., 2016. Deep Learning for Remote Sensing Data: A Technical Tutorial on the State of the Art. *IEEE Geosciences and Remote Sensing Magazine* 4(2): 22-40, doi: 10.1109/MGRS.2016.2540798

Zhu, X.Z., Tuia, D., Mou, L., Xia, G.-S., Zhang, L., Xu, F., and Fraundorfer, F., 2017. Deep Learning in Remote Sensing: A Comprehensive Review and List of Resources. *IEEE Geosciences and Remote Sensing Magazine* 5(4), 8-36, doi: 10.1109/MGRS.2017.2762307.

Appendix A

The following text contains the Python 3 code written by Zackary Leady to implement the deep learning algorithm. As previously mentioned, Zackary Leady left Reclamation prior to producing this report, and no further documentation of this code is available.

```
# -*- coding: utf-8 -*-
"""
Created on Fri May 31 15:22:13 2019

@author: zleady
"""

import os
import sys
import logging
import datetime
import argparse
import rasterio
import numpy as np
import pandas as pd
from scipy.stats import wasserstein_distance
from scipy.spatial import distance
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
import umap
import hdbscan
from plotly.offline import plot
import plotly.graph_objs as go
import plotly.figure_factory as ff


def create_logger(log_file):
    """ Zack's Generic Logger function to create onscreen and file logger

    Parameters
    ----------
    log_file: string
        `log_file` is the string of the absolute filepathname for writing the
        log file too which is a mirror of the onscreen display.

    Returns
    -------
```

```
    logger: logging object

    Notes
    -----
    This function is completely generic and can be used in any python code.
    The handler.setLevel can be adjusted from logging.INFO to any of the other
    options such as DEBUG, ERROR, WARNING in order to restrict what is logged.

    """
    logger = logging.getLogger()
    logger.setLevel(logging.INFO)
# create console handler and set level to info
    handler = logging.StreamHandler()
    handler.setLevel(logging.INFO)
    formatter = logging.Formatter("%(levelname)s - %(message)s")
    handler.setFormatter(formatter)
    logger.addHandler(handler)
# create error file handler and set level to info
    handler = logging.FileHandler(log_file,  "w", encoding=None, delay="true")
    handler.setLevel(logging.INFO)
    formatter = logging.Formatter("%(levelname)s - %(message)s")
    handler.setFormatter(formatter)
    logger.addHandler(handler)
    return logger


def tiffs_in_dir(image_directory):
    img_files = []
    logging.info('Searching for tiffs in: \n {}'.format(image_directory))
    for img_file in os.listdir(image_directory):
        if img_file.endswith(".tif"):
            img_files.append(os.path.join(image_directory, img_file))
    return img_files


def load_image_array(image_path):
    img_obj = rasterio.open(image_path)
    img_arr = img_obj.read()
    logging.info("Image {} has {} indices and {} shape"
                 .format(image_path, img_obj.indexes, img_arr.shape))
    return img_arr


def gen_NDWI(image_path, write_path):
    with rasterio.open(image_path) as dataset:
        green_band2 = dataset.read(2)
```

```
        nir_band4 = dataset.read(4)
        np.seterr(divide='ignore', invalid='ignore')
        ndwi = ((green_band2.astype(float) - nir_band4.astype(float)) /
                (green_band2.astype(float) + nir_band4.astype(float)))
        kwargs = dataset.meta
        kwargs.update(dtype=rasterio.float32, count=1)
        if not os.path.exists(os.path.join(write_path, 'ndwi')):
            os.mkdir(os.path.join(write_path, 'ndwi'))
        output_path = os.path.join(write_path, r"ndwi\{}_ndwi.tif"
                                   .format(os.path.basename(image_path)
                                           .split(".")[0]))
        with rasterio.open(output_path, 'w', **kwargs) as dst:
            dst.write_band(1, ndwi.astype(rasterio.float32))


def gen_NDVI(image_path, write_path):
    with rasterio.open(image_path) as dataset:
        red_band3 = dataset.read(3)
        nir_band4 = dataset.read(4)
        np.seterr(divide='ignore', invalid='ignore')
        ndvi = ((nir_band4.astype(float) - red_band3.astype(float)) /
                (nir_band4.astype(float) + red_band3.astype(float)))
        kwargs = dataset.meta
        kwargs.update(dtype=rasterio.float32, count=1)
        if not os.path.exists(os.path.join(write_path, 'ndvi')):
            os.mkdir(os.path.join(write_path, 'ndvi'))
        output_path = os.path.join(write_path, r"ndvi\{}_ndvi.tif"
                                   .format(os.path.basename(image_path)
                                           .split(".")[0]))
        with rasterio.open(output_path, 'w', **kwargs) as dst:
            dst.write_band(1, ndvi.astype(rasterio.float32))


def create_auxillary_bands(image_files_lst, write_path):
    logging.info('Generating NDWI and NDVI for {} image files'
                 .format(len(image_files_lst)))
    for image_path in image_files_lst:
        gen_NDWI(image_path, write_path)
        gen_NDVI(image_path, write_path)


def gen_class_label_dataset(class_labels_path):
    class_dict = {}
    class_labels_arr = load_image_array(class_labels_path)
    logging.info('Shape of class labels array: {}'
                 .format(class_labels_arr.shape))
```

```
    unique_values, unique_counts = np.unique(class_labels_arr,
                                             return_counts=True)
    logging.info('Unique values of the class_labels_array: {}'
                 .format(unique_values))
    logging.info('Unique counts for each unique value: {}'
                 .format(unique_counts))
    # reshape class_labels_arr to 1D instead of 2D image array
    # class_labels_arr.shape = (1, 1000, 1000)
    class_labels_arr = class_labels_arr.reshape(class_labels_arr.shape[1] *
                                                class_labels_arr.shape[2])
    # new class_labels_arr.shape = (1,000,000;)
    logging.info('Shape of class labels array changed to 1D array: {}'
                 .format(class_labels_arr.shape))
    for i, uv in enumerate(unique_values):
        # indices_unique_value is actually a list of arrays of len() 1 so [0]
        # indices_unique_value[0] is a 1D array of the indices that match uv
        indices_unique_value = np.where(class_labels_arr == uv)
        logging.info('Looping on index i: {} and unique_value uv: {}'
                     .format(i, uv))
        logging.info("Length of indices_unique_value array:" +
                     "{} should match unique_counts: {}"
                     .format(len(indices_unique_value[0]), unique_counts[i]))
        assert len(indices_unique_value[0]) == unique_counts[i]
        class_dict[uv] = [indices_unique_value[0], unique_counts[i]]
    return class_labels_arr, class_dict


def gen_pixel_timeseries(image_files_lst, ndvi_files_lst, ndwi_files_lst):
    date_order = []
    final_array = np.empty((0, 6, 1000, 1000),dtype=np.float32)
    for img_file in image_files_lst:
        img_file_base = os.path.basename(img_file).split(".")[0]
        for ndvi_file in ndvi_files_lst:
            ndvi_file_base = os.path.basename(ndvi_file).split(".")[0]
            if img_file_base in ndvi_file_base:
                for ndwi_file in ndwi_files_lst:
                    ndwi_file_base = os.path.basename(ndwi_file).split(".")[0]
                    if img_file_base in ndwi_file_base:
                        date_order.append(ndwi_file_base.split("_")[0]) # change from Z to V
naming
                        logging.info('Found match img {} == ndvi {} == ndwi {}'
                                     .format(img_file_base, ndvi_file_base,
                                             ndwi_file_base))
                        img_arr = load_image_array(img_file)
                        ndvi_arr = load_image_array(ndvi_file)
                        ndwi_arr = load_image_array(ndwi_file)
```

```
                logging.info("Shape of image: {}; Shape of ndvi: {};"
                        "Shape of ndwi: {}".format(img_arr.shape,
                                                   ndvi_arr.shape,
                                                   ndwi_arr.shape)
                        )
                concat_arr = np.concatenate([img_arr, ndvi_arr,
                                             ndwi_arr])
                logging.info('Concat arr shape: {}'
                        .format(concat_arr.shape))
                concat_arr = concat_arr[np.newaxis, :]
                logging.info('Concat arr new shape: {}'
                        .format(concat_arr.shape))
                if(concat_arr.shape[2] == 1000 and
                   concat_arr.shape[3] == 1000):
                    logging.info('old final array growing shape: {}'
                            .format(final_array.shape))
                    final_array = np.concatenate([final_array,
                                                  concat_arr],
                                                 axis=0)
                    logging.info('new final array growing shape: {}'
                            .format(final_array.shape))
    logging.info('Final array end shape: {}'.format(final_array.shape))
    return final_array, date_order


def gen_main_datasets(image_files_lst, write_path, class_labels_path):
    ndvi_dir = os.path.join(write_path, 'ndvi')
    ndwi_dir = os.path.join(write_path, 'ndwi')
    logging.info('Looking for NDVI files in: \n {}'.format(ndvi_dir))
    logging.info('Looking for NDWI files in: \n {}'.format(ndwi_dir))
    ndvi_files_lst = tiffs_in_dir(ndvi_dir)
    logging.info('Found {} ndvi paths'.format(len(ndvi_files_lst)))
    ndwi_files_lst = tiffs_in_dir(ndwi_dir)
    logging.info('Found {} ndwi paths'.format(len(ndwi_files_lst)))
    assert len(image_files_lst) == len(ndvi_files_lst) == len(ndwi_files_lst)
    class_labels_arr, class_dict = gen_class_label_dataset(class_labels_path)
    pixel_timeseries_arr, date_order = gen_pixel_timeseries(image_files_lst,
                                                ndvi_files_lst,
                                                ndwi_files_lst)
    return class_labels_arr, class_dict, pixel_timeseries_arr, date_order


def create_class_pixel_timeseries_graphs(write_path, date_order,
                                 class_dict, pixel_timeseries_arr):
    # shape of pixel_timeseries_arr should be
    # (# of images, # of bands, x pixels, y pixels)
```

```
pd_date = [pd.to_datetime(x) for x in date_order]
plot_data = []
plot_data_3d = []
for ck in class_dict.keys():
    class_dict_obj = class_dict.get(ck)
    class_indices = np.random.choice(class_dict_obj[0], size=3)
    for indx in class_indices:
        reclaimed_2d_indx = np.unravel_index(indx,
                                    (pixel_timeseries_arr
                                     .shape[2],
                                     pixel_timeseries_arr
                                     .shape[3]))
        ndvi_pixel_graph_timeseries = pixel_timeseries_arr[:, 4,
                                              reclaimed_2d_indx[0],
                                              reclaimed_2d_indx[1]]
        ndwi_pixel_graph_timeseries = pixel_timeseries_arr[:, 5,
                                              reclaimed_2d_indx[0],
                                              reclaimed_2d_indx[1]]
        temp_trace_ndvi = go.Scatter(x=pd_date,
                                y=ndvi_pixel_graph_timeseries,
                                mode="lines",
                                name='{}_{}_ndvi'.format(ck, indx),
                                legendgroup='{}_ndwi'.format(ck))
        temp_trace_ndwi = go.Scatter(x=pd_date,
                                y=ndwi_pixel_graph_timeseries,
                                mode="lines",
                                name='{}_{}_ndwi'.format(ck, indx),
                                legendgroup='{}_ndvi'.format(ck))
        plot_data.append(temp_trace_ndvi)
        plot_data.append(temp_trace_ndwi)
        temp_trace_3d = go.Scatter3d(x=pd_date,
                                y=ndvi_pixel_graph_timeseries,
                                z=ndwi_pixel_graph_timeseries,
                                name='{}_{}'.format(ck, indx))
        plot_data_3d.append(temp_trace_3d)
# 2D
layout = go.Layout()
fig = go.Figure(data=plot_data, layout=layout)
fname = os.path.join(write_path, 'class_pixel_timeseries_ndvi_ndwi.html')
plot(fig, filename=fname, auto_open=False, show_link=False,
    config=dict(displaylogo=False))
# 3D
layout3d = go.Layout()
fig3d = go.Figure(data=plot_data_3d, layout=layout3d)
fname3d = os.path.join(write_path,
                  'class pixel_timeseries_3d_ndvi_ndwi.html')
```

```
    plot(fig3d, filename=fname3d, auto_open=False, show_link=False,
         config=dict(displaylogo=False))
    return


def create_class_banded_graphs(write_path, date_order,
                               class_dict, pixel_timeseries_arr):
    ndvi_plot_data = []
    ndwi_plot_data = []
    for ck in class_dict.keys():
        class_dict_obj = class_dict.get(ck)
        x_2d_indx_lst = []
        y_2d_indx_lst = []
        for indx in class_dict_obj[0]:
            reclaimed_2d_indx = np.unravel_index(indx,
                                                 (pixel_timeseries_arr
                                                  .shape[2],
                                                  pixel_timeseries_arr
                                                  .shape[3]))
            x_2d_indx_lst.append(reclaimed_2d_indx[0])
            y_2d_indx_lst.append(reclaimed_2d_indx[1])
        ck_data = pixel_timeseries_arr[:, :, x_2d_indx_lst,
                                       y_2d_indx_lst]
        for b in [4, 5]:
            max_lst = []
            min_lst = []
            for t in range(pixel_timeseries_arr.shape[0]):
                print(pixel_timeseries_arr.shape)
                print(ck_data.shape)
                max_vec = np.max(ck_data[t, b, :])
                min_vec = np.min(ck_data[t, b, :])
                max_lst.append(max_vec)
                min_lst.append(min_vec)
            if b == 4:
                ndvi_plot_data.append([ck, max_lst, min_lst])
            elif b == 5:
                ndwi_plot_data.append([ck, max_lst, min_lst])
    for aux_band in [['ndvi', ndvi_plot_data], ['ndwi', ndwi_plot_data]]:
        plot_data = []
        for obj in aux_band[1]:
            x = date_order
            x_rev = date_order[::-1]
            y1_upper = obj[1]
            y1_lower = obj[2]
            trace_temp = go.Scatter(x=x+x_rev, y=y1_upper+y1_lower,
                                    fill='tozerox',
```

```
                        showlegend=True,
                        name='{}'.format(obj[0]))
            plot_data.append(trace_temp)
        layout = go.Layout()
        fig = go.Figure(data=plot_data, layout=layout)
        fname = os.path.join(write_path, '{}_class_banded.html'
                             .format(aux_band[0]))
        plot(fig, filename=fname, auto_open=False, show_link=False,
             config=dict(displaylogo=False))
    return


def create_class_2d_density_graphs(write_path, date_order,
                                   class_dict, pixel_timeseries_arr):
    pd_date = [pd.to_datetime(x) for x in date_order]
    assert len(pd_date) == len(range(pixel_timeseries_arr.shape[0]))
    for t, dt in zip(range(pixel_timeseries_arr.shape[0]), pd_date):
        ndvi = pixel_timeseries_arr[t, 4, :, :]
        ndwi = pixel_timeseries_arr[t, 5, :, :]
        ndvi_plot = ndvi.reshape(ndvi.shape[0]*ndvi.shape[1])
        ndwi_plot = ndwi.reshape(ndwi.shape[0]*ndwi.shape[1])
        print(ndvi.shape, ndwi.shape)
        plot_data = []

        for ck in class_dict.keys():
            if not ck == 0:
                class_dict_obj = class_dict.get(ck)
                x_2d_indx_lst = []
                y_2d_indx_lst = []
                for indx in class_dict_obj[0]:
                    reclaimed_2d_indx = np.unravel_index(indx,
                                                         (pixel_timeseries_arr
                                                          .shape[2],
                                                          pixel_timeseries_arr
                                                          .shape[3]))
                    x_2d_indx_lst.append(reclaimed_2d_indx[0])
                    y_2d_indx_lst.append(reclaimed_2d_indx[1])
                x0 = [x for x in ndvi[x_2d_indx_lst, y_2d_indx_lst]]
                y0 = [x for x in ndwi[x_2d_indx_lst, y_2d_indx_lst]]
                print(len(x0), x0[0], len(y0), y0[0])
                trace0 = go.Scattergl(x=x0, y=y0, mode='markers',
                                      name='c{}'.format(ck),
                                      marker=dict(size=2, opacity=1.0),
                                      showlegend=True)
                plot_data.append(trace0)
        trace1 = go.Scattergl(x=ndvi_plot, y=ndwi_plot, mode='markers', name='points',
```

```
                    marker=dict(color='rgb(0,0,100)',
                                size=1, opacity=0.4),
                    showlegend=False)
    plot_data.append(trace1)
    trace2 = go.Histogram2dContour(x=ndvi_plot, y=ndwi_plot, name='density',
                        ncontours=20, colorscale='Hot',
                        reversescale=True, showscale=False,
                        showlegend=False)
    plot_data.append(trace2)
    trace3 = go.Histogram(x=ndvi_plot, name='x density',
                    marker=dict(color='rgb(102,0,0)'),
                    showlegend=False,
                    yaxis='y2')
    plot_data.append(trace3)
    trace4 = go.Histogram(y=ndwi_plot, name='y density',
                    marker=dict(color='rgb(102,0,0)'),
                    showlegend=False,
                    xaxis='x2')
    plot_data.append(trace4)
    layout = go.Layout(showlegend=True, autosize=False,
                    width=800, height=750,
                    xaxis=dict(domain=[0, 0.85], showgrid=False,
                            zeroline=False),
                    yaxis=dict(domain=[0, 0.85], showgrid=False,
                            zeroline=False),
                    margin=dict(t=50),
                    hovermode='closest',
                    bargap=0,
                    xaxis2=dict(domain=[0.85, 1], showgrid=False,
                            zeroline=False),
                    yaxis2=dict(domain=[0.85, 1], showgrid=False,
                            zeroline=False))
    fig = go.Figure(data=plot_data, layout=layout)
    fname = os.path.join(write_path,
                    'dt{}_t{}_class_density2d.html'
                    .format(dt.date(), t))
    plot(fig, filename=fname, auto_open=False, show_link=False,
        config=dict(displaylogo=False))
return


def create_class_distance_heatmaps(write_path, class_dict,
                            pixel_timeseries_arr):
    print(pixel_timeseries_arr.shape)
    reshaped_pixel_arr = pixel_timeseries_arr.reshape([pixel_timeseries_arr
                                        .shape[0],
```

```
                                        pixel_timeseries_arr
                                        .shape[1],
                                        pixel_timeseries_arr
                                        .shape[2] *
                                        pixel_timeseries_arr
                                        .shape[3]])
    input_dist_ck_arr_lst = []
    input_dist_arr_lst = []
    for ck in sorted(class_dict.keys()):
        if not ck == 0:
            class_indices = class_dict.get(ck)
            class_data = reshaped_pixel_arr[:, :, class_indices[0]]
            print(ck, class_data.shape)
            class_vector_data = class_data.reshape([class_data.shape[2],
                                        class_data.shape[0] *
                                        class_data.shape[1]])
            print(class_vector_data[0:10, :])
            print(class_vector_data.shape)
            input_dist_ck_arr_lst.append([ck, class_vector_data])
            input_dist_arr_lst.append(class_vector_data)
    import itertools
    iter_class_combinations = list(itertools
                                .combinations([x[0] for x in
                                        input_dist_ck_arr_lst], 2))
    print(iter_class_combinations)
    distance_metric_lst = ['euclidean', 'correlation', 'cosine', 'mahalanobis',
                           'wasserstein']
    for dist in distance_metric_lst:
        if dist == 'wasserstein':
            pass
#           for combo in iter_class_combinations:
#               u_indx = combo[0]-1
#               v_indx = combo[1]-1
#               print(combo, u_indx, v_indx)
#               u = input_dist_ck_arr_lst[u_indx][1]
#               v = input_dist_ck_arr_lst[v_indx][1]
#               calculated_dist = wasserstein_distance(u, v)
#               print(calculated_dist)
        else:
            X_arr = np.vstack(input_dist_arr_lst)
            print(dist, X_arr.shape)
            calculated_dist = distance.pdist(X_arr, metric=dist)
            print(dist, calculated_dist.shape)
            square_dist = distance.squareform(calculated_dist)
            print(dist, square_dist.shape)
            data = []
```

```
        trace1 = go.Heatmap(z=square_dist)
        data.append(trace1)
        layout = go.Layout()
        fig = go.Figure(data=data, layout=layout)
        fname = os.path.join(write_path,
                             '{}_class_distance_heatmap.html'.format(dist))
        plot(fig, filename=fname, auto_open=False, show_link=False,
            config=dict(displaylogo=False))
    return


def create_direct_comparison_class_distance_heatmaps(write_path, class_dict,
                                                     pixel_timeseries_arr):
    distance_metric_lst = ['euclidean', 'correlation', 'cosine', 'mahalanobis',
                           'wasserstein']
    import itertools
    iter_class_combinations = list(itertools
                                   .combinations([x for x in
                                                  sorted(class_dict.keys())
                                                  if not x == 0],
                                                 2))
    print(iter_class_combinations)
    reshaped_pixel_arr = pixel_timeseries_arr.reshape([pixel_timeseries_arr
                                                       .shape[2] *
                                                       pixel_timeseries_arr
                                                       .shape[3],
                                                       pixel_timeseries_arr
                                                       .shape[0] *
                                                       pixel_timeseries_arr
                                                       .shape[1]])
    print(reshaped_pixel_arr.shape)
    print(reshaped_pixel_arr[0:10, :])
    for class_combo in iter_class_combinations:
        print(class_combo)
        class0_data_dict = class_dict.get(class_combo[0])
        class1_data_dict = class_dict.get(class_combo[1])
        class0_indices = class0_data_dict[0]
        class1_indices = class1_data_dict[0]
        class0_data_arr = reshaped_pixel_arr[class0_indices, :]
        class1_data_arr = reshaped_pixel_arr[class1_indices, :]
        logging.info('class0: {} = {} = {}'
                     .format(class_combo[0], class0_data_dict[1],
                             class0_data_arr.shape))
        logging.info('class1: {} = {} = {}'
                     .format(class_combo[1], class1_data_dict[1],
                             class1_data_arr.shape))
```

```
        for dist in distance_metric_lst:
            if dist == 'wasserstein':
                calculated_dist = distance.cdist(class0_data_arr,
                                                 class1_data_arr,
                                                 metric=lambda u, v: wasserstein_distance(u, v))
            else:
                calculated_dist = distance.cdist(class0_data_arr,
                                                 class1_data_arr, metric=dist)
            logging.info('{}, {}'.format(dist, calculated_dist.shape))
            data = []
            trace1 = go.Heatmap(z=calculated_dist)
            data.append(trace1)
            layout = go.Layout()
            fig = go.Figure(data=data, layout=layout)
            fname = os.path.join(write_path,
                                 '{}_{}_{}_class_compare_dist_heatmap.html'
                                 .format(dist, class_combo[0],
                                         class_combo[1]))
            plot(fig, filename=fname, auto_open=False, show_link=False,
                 config=dict(displaylogo=False))
    return


def create_class_distance_distplots(write_path, class_dict,
                                    pixel_timeseries_arr):
    distance_metric_lst = ['chebyshev','cityblock','correlation', 'euclidean', 'seuclidean',
'cosine', 'mahalanobis', # 'correlation',
                           'wasserstein']
    import itertools
    iter_class_combinations = list(itertools
                                   .combinations([x for x in
                                                  sorted(class_dict.keys())
                                                  if not x == 0],
                                                 2))
    print(iter_class_combinations)
    print(pixel_timeseries_arr.shape)
#     pixel_timeseries_arr = pixel_timeseries_arr[:,4:6,:,:]
#     print(pixel_timeseries_arr.shape)
#     print(pixel_timeseries_arr[0,:,0:3,0:3])
#     print(pixel_timeseries_arr.max())
#     print(pixel_timeseries_arr.min())
    reshaped_pixel_arr = pixel_timeseries_arr.reshape([pixel_timeseries_arr
                                                       .shape[2] *
                                                       pixel_timeseries_arr
                                                       .shape[3],
                                                       pixel_timeseries_arr
```

```
                                        .shape[0] *
                                        pixel_timeseries_arr
                                        .shape[1]])
    print(reshaped_pixel_arr.shape)
    print(reshaped_pixel_arr[0:10, :])
    for dist in distance_metric_lst:
        hist_data = []
        group_labels = []
        for class_combo in iter_class_combinations:
            print(class_combo)
            class0_data_dict = class_dict.get(class_combo[0])
            class1_data_dict = class_dict.get(class_combo[1])
            class0_indices = class0_data_dict[0]
            class1_indices = class1_data_dict[0]
            class0_data_arr = reshaped_pixel_arr[class0_indices, :]
            class1_data_arr = reshaped_pixel_arr[class1_indices, :]
            logging.info('class0: {} = {} = {}'
                         .format(class_combo[0], class0_data_dict[1],
                                 class0_data_arr.shape))
            logging.info('class1: {} = {} = {}'
                         .format(class_combo[1], class1_data_dict[1],
                                 class1_data_arr.shape))
            if dist == 'wasserstein':
                calculated_dist = distance.cdist(class0_data_arr,
                                                 class1_data_arr,
                                                 metric=lambda u, v:
                                                       wasserstein_distance(u,v))
            else:
                calculated_dist = distance.cdist(class0_data_arr,
                                                 class1_data_arr, metric=dist)
            logging.info('{}, {}'.format(dist, calculated_dist.shape))
            hist_dist = calculated_dist.flatten()
            logging.info('calculated dist shape: {} \n flattened shape: {}'
                         .format(calculated_dist.shape, hist_dist.shape))
            hist_dist_lst = [x for x in hist_dist]
            hist_data.append(hist_dist_lst)
            group_labels.append('c{}_c{}'.format(class_combo[0], class_combo[1]))
        fig = ff.create_distplot(hist_data, group_labels, show_hist=False, show_rug=False)
        fname = os.path.join(write_path,
                             '{}_class_compare_dist_distplot.html'.format(dist))
        plot(fig, filename=fname, auto_open=False, show_link=False,
             config=dict(displaylogo=False))
    return


def exploratory_graph_analysis(class_labels_arr, class_dict,
```

```
                        pixel_timeseries_arr, date_order, write_path):
#   create_class_pixel_timeseries_graphs(write_path, date_order,
#                                        class_dict, pixel_timeseries_arr)
#   create_class_2d_density_graphs(write_path, date_order,
#                                  class_dict, pixel_timeseries_arr)
#   create_class_distance_heatmaps(write_path, class_dict,
#                                  pixel_timeseries_arr)
#   create_direct_comparison_class_distance_heatmaps(write_path, class_dict,
#                                                    pixel_timeseries_arr)
#   create_class_distance_distplots(write_path, class_dict,
#                                   pixel_timeseries_arr)
    create_class_banded_graphs(write_path, date_order,
                               class_dict, pixel_timeseries_arr)
    return


def gen_generic_embedding3_graph(embedding_arr, labels, title, file_name):
    print(embedding_arr.shape)
    print(labels.shape)
    x_arr = embedding_arr[:, 0]
    y_arr = embedding_arr[:, 1]
    z_arr = embedding_arr[:, 2]
    print(x_arr.shape, y_arr.shape, z_arr.shape)
    embed_trace0 = go.Scatter3d(x=x_arr,
                                y=y_arr,
                                z=z_arr,
                                mode='markers',
                                marker=dict(size=2, color=labels,
                                            colorscale='Viridis',
                                            opacity=0.8),
                                text=labels, hoverinfo='text')
    plot_data = [embed_trace0]
    layout = go.Layout(title=title)
    fig = go.Figure(data=plot_data, layout=layout)
    plot(fig, filename=file_name, auto_open=False, show_link=False,
         config=dict(displaylogo=False))
    return


def gen_generic_embedding2_graph(embedding_arr, labels, title, file_name):
    x_arr = embedding_arr[:, 0]
    y_arr = embedding_arr[:, 1]
    plot_data = []
    for u in np.unique(labels):
        print(u)
        indx_u = np.where(labels == u)
```

```
#       print(indx_u[0])
        print(indx_u[0].shape)
        x_u_arr = x_arr[indx_u[0]]
        print(x_u_arr.shape)
        y_u_arr = y_arr[indx_u[0]]
        print(y_u_arr.shape)
        if not u == 0:
            embed_trace = go.Scattergl(x=x_u_arr,
                                       y=y_u_arr,
                                       mode='markers',
                                       marker=dict(size=5,
                                                   opacity=1.0),
                                       name='c{}'.format(u))
        else:
            embed_trace = go.Scattergl(x=x_u_arr,
                                       y=y_u_arr,
                                       mode='markers',
                                       marker=dict(size=2,
                                                   opacity=0.6),
                                       name='c{}'.format(u))
        plot_data.append(embed_trace)
#    plot_data = [embed_trace0]
    layout = go.Layout(title=title)
    fig = go.Figure(data=plot_data, layout=layout)
    plot(fig, filename=file_name, auto_open=False, show_link=False,
         config=dict(displaylogo=False))
    return

def create_raw_data_embedding_pca(pixel_timeseries_arr, class_labels_arr,
                                  class_dict, write_path, dim=3,
                                  include_labels=True):
    print(pixel_timeseries_arr.shape)
    X = pixel_timeseries_arr.reshape([pixel_timeseries_arr.shape[2] *
                                      pixel_timeseries_arr.shape[3],
                                      pixel_timeseries_arr.shape[0] *
                                      pixel_timeseries_arr.shape[1]])
    print(X.shape)
    pca = PCA(n_components=dim)
    solved_pca = pca.fit(X)
    pca_arr = solved_pca.transform(X)
    print(pca_arr)
    print(pca_arr.shape)
    print(solved_pca.explained_variance_ratio_)
#    print(solved_pca.singular_values_)
#    print(solved_pca.components_)
    if include_labels is True:
```

```
        labels = class_labels_arr.reshape([class_labels_arr.shape[0]])
    else:
        labels = np.zeros([X.shape[0]])
    pca_title = 'Raw PCA Embedding with Dim {}'.format(dim)
    pca_filename = os.path.join(write_path,
                                'raw_data_pca_embedding_dim{}.html'
                                .format(dim))
    gen_generic_embedding2_graph(pca_arr, labels, pca_title, pca_filename)
    return


def create_raw_data_embedding_tsne(pixel_timeseries_arr, class_labels_arr,
                                   class_dict, write_path, dim=3,
                                   include_labels=True):
    print(pixel_timeseries_arr.shape)
    X = pixel_timeseries_arr.reshape([pixel_timeseries_arr.shape[2] *
                                      pixel_timeseries_arr.shape[3],
                                      pixel_timeseries_arr.shape[0] *
                                      pixel_timeseries_arr.shape[1]])
    print(X.shape)
    tsne = TSNE(n_components=dim)
    solved_tsne = tsne.fit(X)
    print(solved_tsne.kl_divergence_)
    print(solved_tsne.n_iter_)
    tsne_arr = solved_tsne.transform(X)
    print(tsne_arr)
    print(tsne_arr.shape)
    if include_labels is True:
        labels = class_labels_arr.reshape([class_labels_arr.shape[0], 1])
    else:
        labels = np.zeros([X.shape[0], 1])
    tsne_title = 'Raw TSNE Embedding with Dim={}'.format(dim)
    tsne_filename = os.path.join(write_path,
                                 'raw_data_tsne_embedding_dim{}.html'
                                 .format(dim))
    gen_generic_embedding3_graph(tsne_arr, labels, tsne_title, tsne_filename)
    return


def create_raw_data_embedding_umap(pixel_timeseries_arr, class_labels_arr,
                                   class_dict, write_path, dim=3,
                                   include_labels=True):
    print(pixel_timeseries_arr.shape)
    X = pixel_timeseries_arr.reshape([pixel_timeseries_arr.shape[2] *
                                      pixel_timeseries_arr.shape[3],
                                      pixel_timeseries_arr.shape[0] *
```

```
                                pixel_timeseries_arr.shape[1]])
    print(X.shape)
    reducer = umap.UMAP(n_components=dim)
    solved_umap = reducer.fit(X)
    umap_arr = solved_umap.transform(X)
    print(umap_arr)
    print(umap_arr.shape)
    if include_labels is True:
        labels = class_labels_arr.reshape([class_labels_arr.shape[0]])
    else:
        labels = np.zeros([X.shape[0]])
    umap_title = 'Raw UMAP Embedding with Dim={}'.format(dim)
    umap_filename = os.path.join(write_path,
                                 'raw_data_umap_embedding_dim{}.html'
                                 .format(dim))
    gen_generic_embedding2_graph(umap_arr, labels, umap_title, umap_filename)
    return


def create_analysis_embeddings(pixel_timeseries_arr, class_labels_arr,
                               class_dict, write_path, dim=3,
                               include_labels=True, metric=None):
    if metric == 'wasserstein':
        reducer = umap.UMAP(n_components=dim,
                            metric=lambda u, v: wasserstein_distance(u, v))
        pca = PCA(n_components=2,
                  metric=lambda u, v: wasserstein_distance(u, v))
    else:
        reducer = umap.UMAP(n_components=dim, metric=metric)
        pca = PCA(n_components=2, metric=metric)
    X = pixel_timeseries_arr.reshape([pixel_timeseries_arr.shape[2] *
                                      pixel_timeseries_arr.shape[3],
                                      pixel_timeseries_arr.shape[0] *
                                      pixel_timeseries_arr.shape[1]])
    if include_labels is True:
        labels = class_labels_arr.reshape([class_labels_arr.shape[0], 1])
    else:
        labels = np.zeros([X.shape[0], 1])
    solved_umap = reducer.fit(X)
    umap_arr = solved_umap.transform(X)
    print(umap_arr)
    print(umap_arr.shape)
    umap_title = 'Analysis UMAP Embedding with Dim={}'.format(dim)
    umap_filename = os.path.join(write_path,
                                 'analysis_umap_embedding_dim{}'.format(dim))
    gen_generic_embedding3_graph(umap_arr, labels, umap_title, umap_filename)
```

```
    solved_pca = pca.fit(X)
    pca_arr = solved_pca.transform(X)
    print(pca_arr)
    print(pca_arr.shape)
    print(solved_pca.explained_variance_ratio_)
    pca_title = 'Analysis PCA Embedding with Dim={}'.format(2)
    pca_filename = os.path.join(write_path,
                            'analysis_pca_embedding_dim{}'.format(2))
    gen_generic_embedding3_graph(pca_arr, labels, pca_title, pca_filename)
    return pca_arr, umap_arr


def kmeans_analysis(X, raw_E, class_labels_arr, class_dict,
                    write_path, k_lst=[],
                    include_labels=True, dim=3, metric=None):
    if include_labels is True:
        labels = class_labels_arr.reshape([class_labels_arr.shape[0], 1])
    else:
        labels = np.zeros([X.shape[0], 1])
    for k in k_lst:
        kmeans = KMeans(n_clusters=k).fit(X)
        klabels = kmeans.labels_
        logging.info('Finished kmeans for k={} and metric={}: and shape={}'
                     .format(k, metric, klabels.shape))
        plot_data = []
        cluster_trace = go.Scatter3d(x=raw_E[:, 0],
                                     y=raw_E[:, 1],
                                     z=raw_E[:, 2],
                                     mode='markers',
                                     marker=dict(size=2, color=klabels,
                                                 colorscale='Viridis',
                                                 colorbar=dict(
                                                     title='Cluster'),
                                                 opacity=0.8),
                                     text=klabels, hoverinfo='text')
        if include_labels is True:
            for ck in sorted(class_dict.keys()):
                if not ck == 0:
                    class_indices = class_dict.get(ck)[0]
                    temp_class_arr = raw_E[class_indices, :]
                    clabels = labels[class_indices, :]
                    temp_class_trace = go.Scatter3d(x=temp_class_arr[:, 0],
                                                    y=temp_class_arr[:, 1],
                                                    z=temp_class_arr[:, 2],
                                                    mode='markers',
                                                    marker=dict(size=2,
```

```
                                                opacity=1.0),
                                        text=clabels,
                                        hoverinfo='text')
                plot_data.append(temp_class_trace)
        plot_data.append(cluster_trace)
        cluster_title = ("Cluster Kmeans Embedding from Raw" +
                    "Embedding of K={} Metric={}".format(k, metric))
        layout = go.Layout(title=cluster_title)
        fig = go.Figure(data=plot_data, layout=layout)
        fname = os.path.join(write_path,
                         "kmeans_embedding_{}as3_k{}_{}.html"
                         .format(dim, k, metric))
        plot(fig, filename=fname, auto_open=False, show_link=False,
             config=dict(displaylogo=False))
        wklabels = klabels.reshape(1000, 1000)
        logging.info('wklabels write out shape: {}'.format(wklabels.shape))
        with rasterio.open(write_path, 'ZRedImages',
                        'T20161011_223732_3_1_0c54.tif') as img_data:
            img_data.read()
            print(img_data.indexes)
            kwargs = img_data.meta
            kwargs.update(dtype=rasterio.float32, count=1)
            with rasterio.open(os.path.join(write_path,
                                        "klabels_d{}_k{}_{}"
                                        .format(dim, k,
                                                metric))) as kdst:
                kdst.write_band(1, wklabels.astype(rasterio.float32))
            kdst.close()
        img_data.close()
    return klabels


def hdbscan_analysis(X, raw_E, class_labels_arr, write_path,
                     class_dict,
                     min_cluster_size_lst=[],
                     min_samples_lst=[],
                     include_labels=True, dim=3, metric=None):
    if include_labels is True:
        labels = class_labels_arr.reshape([class_labels_arr.shape[0], 1])
    else:
        labels = np.zeros([X.shape[0], 1])
    for mc in min_cluster_size_lst:
        for ms in min_samples_lst:
            #  cluster_selection_method='leaf'
            clusterer = hdbscan.HDBSCAN(min_cluster_size=mc,
                                        min_samples=ms,
```

```
                        metric='euclidean').fit(X)
logging.info("""Clustering for min_cluster={},
            min_samples={}, and metric={} has label shape: {}"""
            .format(ms, mc, metric, clusterer.labels_.shape))
hlabels = clusterer.labels_
n_c = clusterer.labels_.max()
logging.info("Number of clusters (add 1) : {}".format(n_c))
plot_data = []
cluster_trace = go.Scatter3d(x=raw_E[:, 0],
                        y=raw_E[:, 1],
                        z=raw_E[:, 2],
                        mode='markers',
                        marker=dict(size=2, color=hlabels,
                                colorscale='Viridis',
                                colorbar=dict(
                                      title='Cluster'),
                                opacity=0.8),
                        text=hlabels, hoverinfo='text')
if include_labels is True:
    for ck in sorted(class_dict.keys()):
        if not ck == 0:
            class_indices = class_dict.get(ck)[0]
            temp_class_arr = raw_E[class_indices, :]
            clabels = labels[class_indices, :]
            temp_class_trace = go.Scatter3d(x=temp_class_arr[:, 0],
                                    y=temp_class_arr[:, 1],
                                    z=temp_class_arr[:, 2],
                                    mode='markers',
                                    marker=dict(size=2,
                                              opacity=1.0),
                                    text=clabels,
                                    hoverinfo='text')
            plot_data.append(temp_class_trace)
plot_data.append(cluster_trace)
cluster_title = ("Cluster HDBSCAN Embedding from Raw" +
                "Embedding of MC-MS={}-{} Metric={}"
                .format(mc, ms, metric))
layout = go.Layout(title=cluster_title)
fig = go.Figure(data=plot_data, layout=layout)
fname = os.path.join(write_path,
                    "hdbscan_embedding_{}as3_mc{}_ms{}_{}.html"
                    .format(dim, mc, ms, metric))
plot(fig, filename=fname, auto_open=False, show_link=False,
     config=dict(displaylogo=False))
whlabels = hlabels.reshape(1000, 1000)
logging.info('whlabels write out shape: {}'.format(whlabels.shape))
```

```
            with rasterio.open(write_path, 'ZRedImages',
                               'T20161011_223732_3_1_0c54.tif') as img_data:
                img_data.read()
                print(img_data.indexes)
                kwargs = img_data.meta
                kwargs.update(dtype=rasterio.float32, count=1)
                with rasterio.open(os.path.join(write_path,
                                                "hlabels_d{}_mc{}_ms{}_{}"
                                                .format(dim, mc, ms,
                                                        metric))) as kdst:
                    kdst.write_band(1, whlabels.astype(rasterio.float32))
                kdst.close()
            img_data.close()
        return hlabels


def exploratory_embedding_analysis(class_labels_arr, class_dict,
                                   pixel_timeseries_arr, date_order,
                                   write_path):
    raw_pca = create_raw_data_embedding_pca(pixel_timeseries_arr,
                                            class_labels_arr,
                                            class_dict, write_path, dim=2,
                                            include_labels=True)
#     sys.exit(0)
#     create_raw_data_embedding_tsne(pixel_timeseries_arr, class_labels_arr,
#                                    class_dict, write_path, dim=3,
#                                    include_labels=True)
    raw_umap = create_raw_data_embedding_umap(pixel_timeseries_arr,
                                              class_labels_arr,
                                              class_dict, write_path, dim=2,
                                              include_labels=True)
    sys.exit(0)
    k_lst = [5, 7, 10, 15, 20]
    m_lst = ['chebyshev', 'cityblock', 'correlation', 'euclidean',
             'seuclidean', 'cosine', 'mahalanobis',
             'wasserstein']
    # 'correlation',
    for d in [30]:
        for m in m_lst:
            pca_arr, umap_arr = create_analysis_embeddings(pixel_timeseries_arr,
                                                           class_labels_arr,
                                                           class_dict,
                                                           write_path, dim=d,
                                                           include_labels=True,
                                                           metric=m)
            for X in [['pca', pca_arr, raw_pca], ['umap', umap_arr, raw_umap]]:
```

```
            print(X[0])
            # all clusterings in euclidean based on pca/umap reduction
            kmeans_arr = kmeans_analysis(X[1], X[2], class_labels_arr,
                                         class_dict,
                                         write_path, k_lst=k_lst,
                                         metric=m,
                                         include_labels=True, dim=d)
            #  rkmeans_arr = recombinator_kmeans_analysis(X[1])
            #  kmedoids_arr = kmedoids_analysis(X[1], k=[])
            hdbscan_arr = hdbscan_analysis(X[1],
                                           min_cluster_size=[],
                                           min_samples=[])
    return


def standardize_data(X_arr, write_path, scaler_type='minmax'):
    logging.warning('Note: scalers work on np.array columns not rows')
    reshaped_scaler_arr = X_arr.reshape([X_arr.shape[2] *
                                         X_arr.shape[3],
                                         X_arr.shape[0] *
                                         X_arr.shape[1]])
    print(X_arr.shape)
    print(reshaped_scaler_arr.shape)
    df_stats = pd.DataFrame.from_records(reshaped_scaler_arr)
    print(df_stats.shape)
    df_describe = df_stats.describe()
    print(df_describe.shape)
    pre_fname = os.path.join(write_path,
                             'FeatureStats_{}.csv'.format(scaler_type))
    df_describe.to_csv(pre_fname, sep=",")
    if scaler_type == 'minmax':
        scaler = MinMaxScaler(feature_range=(0, 1))
    elif scaler_type == 'robust':
        scaler = RobustScaler()
    elif scaler_type == 'standard':
        scaler = StandardScaler()
    scaler_obj = scaler.fit(reshaped_scaler_arr)
    scaled_data_arr = scaler_obj.transform(reshaped_scaler_arr)
    df_scaled_stats = pd.DataFrame.from_records(scaled_data_arr)
    df_scaled_describe = df_scaled_stats.describe()
    post_fname = os.path.join(write_path,
                              'FeatureScaledStats_{}.csv'.format(scaler_type))
    df_scaled_describe.to_csv(post_fname, sep=",")
    print(scaled_data_arr.shape)
    scaled_data_arr = np.nan_to_num(scaled_data_arr)
    reshaped_output_arr = scaled_data_arr.reshape([X_arr.shape[0],
```

```
                                        X_arr.shape[1],
                                        X_arr.shape[2],
                                        X_arr.shape[3]])
    print(reshaped_output_arr.shape)
    return reshaped_output_arr


if __name__ == "__main__":
    # begin runtime clock
    start = datetime.datetime.now()
    # determine the absolute file pathname of this *.py file
    abspath = os.path.abspath(__file__)
    # from the absolute file pathname determined above,
    # extract the directory path
    dir_name = os.path.dirname(abspath)
    # initiate logger
    log_file = os.path.join(dir_name, 'ST1867_analyzer9000_{}.log'
                            .format(start.date()))
    create_logger(log_file)
    # create the command line parser object from argparse
    parser = argparse.ArgumentParser()
    # set the command line arguments available to user's
    parser.add_argument("--image_directory", "-imdr", type=str,
                        help="Provide the absolute folder name containing PS\
                        images")
    parser.add_argument("--class_labels_tif", "-cltif", type=str,
                        help="Provide the absolute file path to the class\
                        labels tiff")
    parser.add_argument("--write", "-w", type=str,
                        help="Provide the absolute folder name for writing\
                        outputs, i.e. project directory")
    parser.add_argument("--standardize", "-s", default=False,
                        action='store_true',
                        help="Provide True or False for standardizing\
                        the pixel data")
    # create an object of the command line inputs
    args = parser.parse_args()
    # read the command line inputs into a Python dictionary
    ini_dict = vars(args)
    img_files_lst = tiffs_in_dir(ini_dict.get("image_directory"))
    logging.info('Found total of {} images for processing'
                 .format(len(img_files_lst)))
    create_auxillary_bands(img_files_lst, ini_dict.get("write"))
    (class_labels_arr, class_dict, pixel_timeseries_arr,
     date_order) = gen_main_datasets(img_files_lst, ini_dict.get("write"),
                                     ini_dict.get("class_labels_tif"))
```

```
    if ini_dict.get("standardize") is True:
#       for st in ['minmax', 'robust', 'standard']:
          pixel_timeseries_arr = standardize_data(pixel_timeseries_arr,
                                                  ini_dict.get("write"),
                                                  scaler_type='robust')
#       sys.exit(0)
    else:
        logging.warning('You are proceeding with UNSTANDARIZED DATA')
        ans = input("Are you sure you want to proceed?[Y/N]")
        if 'N' in ans or 'n' in ans:
            sys.exit(0)
#    exploratory_graph_analysis(class_labels_arr, class_dict,
#                               pixel_timeseries_arr, date_order,
#                               ini_dict.get("write"))
    exploratory_embedding_analysis(class_labels_arr, class_dict,
                                   pixel_timeseries_arr, date_order,
                                   ini_dict.get("write"))
    elapsed_time = datetime.datetime.now() - start
    logging.info('Runtime: {}'.format(elapsed_time))
```

Appendix B

The following pages contain slides from a presentation that was given by Zackary Leady to a group of Reclamation Science and Technology researchers on March 30, 2020.

BUREAU OF RECLAMATION

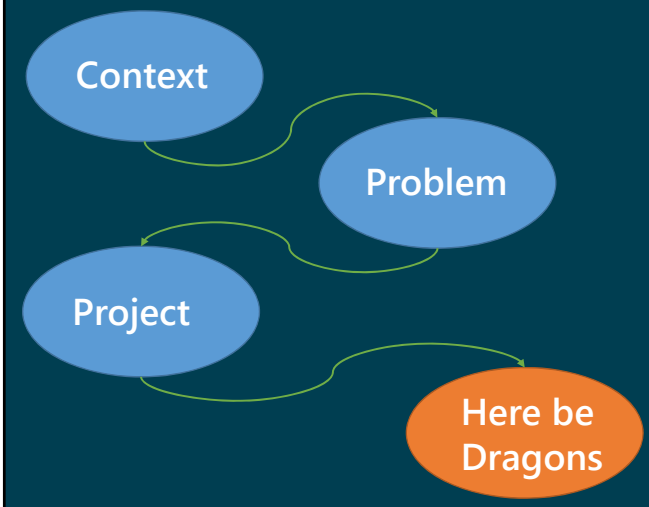
S&T 1867: Temporary Floodplain Delineation using High Resolution Satellite Imagery and Machine Learning

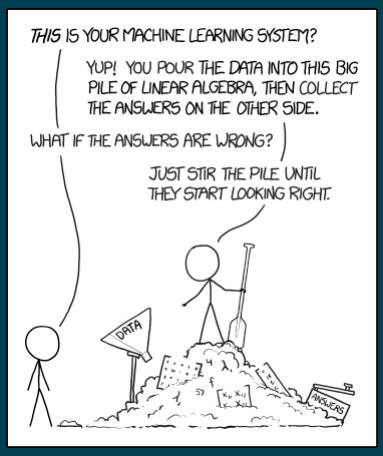
Preliminary Results – Making the Impossible, Possible

By: Zackary Leady

03/30/2020

Roadmap





Contextual Prelude

- Central Valley Improvement Act (CVPIA)
  - Mandate to double salmon population
- Development of DSM models for Salmonids
  - Need for habitat delineation values (acres)
- Temporary Floodplain is an excellent habitat for faster growth
  - Consumnes River study

The Effect of Temporary Floodplain Habitat

Jeffres, C. A., Opperman, J. J., & Moyle, P. B. (2008). Ephemeral floodplain habitats provide best growth conditions for juvenile Chinook salmon in a California river. *Environmental Biology of Fishes*, *83*(4), 449-458. (54 days)

A Problem: How to Delineate?

- Traditional method
  - Survey crew – millions of dollars (Trinity River)
- Remote Sensing Traditional method
  - Normalized Difference Water Index (NDWI)
- Remote Sensing with Machine Learning
  - Unsupervised or Supervised

A Project

S&T 1867: Temporary Floodplain Delineation using High Resolution Remote Sensing and Machine Learning

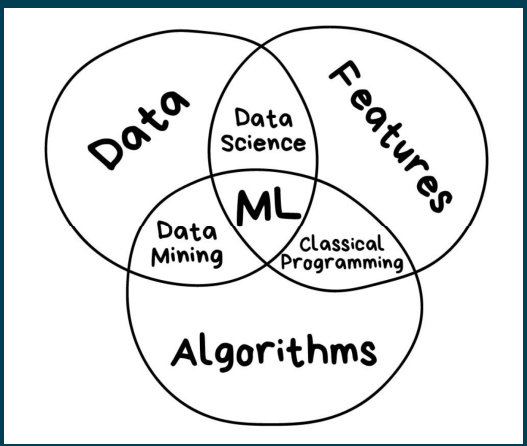
- Requirements:
  - High Temporal and Spatial Resolution Data
  - Automated geospatial toolkit
  - Automated delineation toolkit
  - Open Source / User Friendly

A Problem Revisited: How to Delineate?

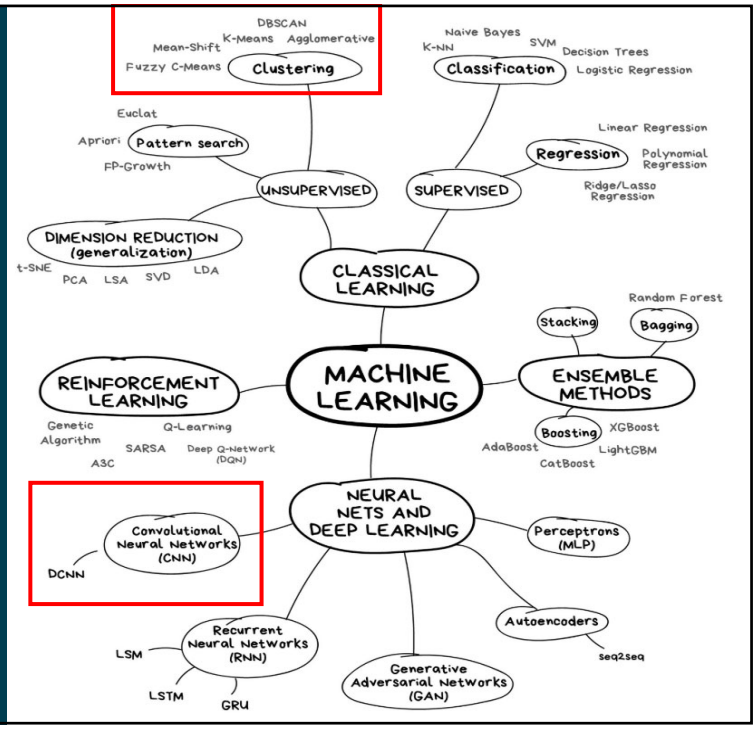
- High Temporal and Spatial Resolution
  - Planet data, Sentinel-2
- Automated Delineation
  - Machine/Deep learning (computer vision)
- Open Source / User Friendly
  - Python

What is Machine Learning?

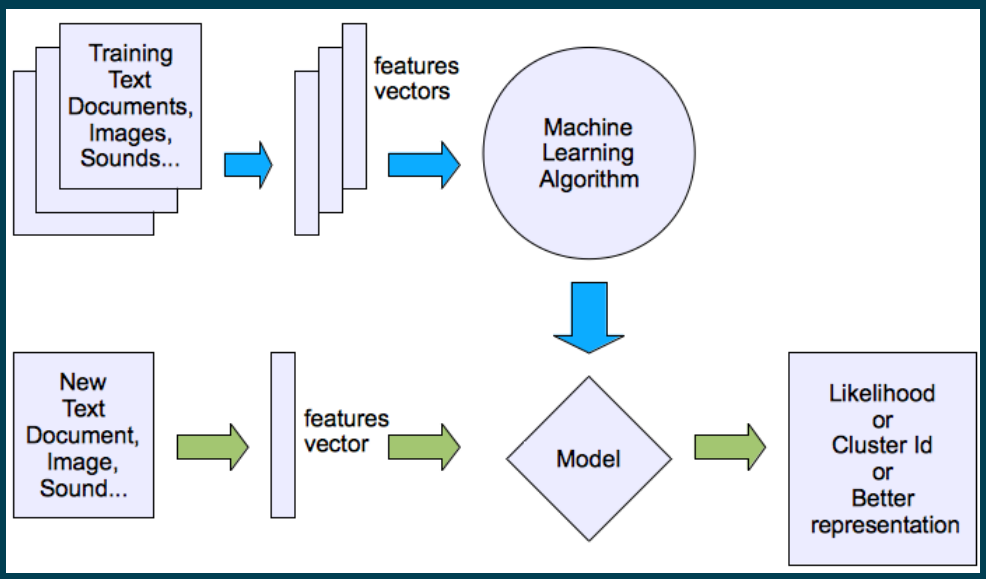
- Machine Learning is a branch of AI
  - Focuses on allowing computer to learn patterns without being explicitly programmed
- Unsupervised vs. Supervised
  - Unsupervised – no labels
  - Supervised – labels



Map of the Machine Learning World

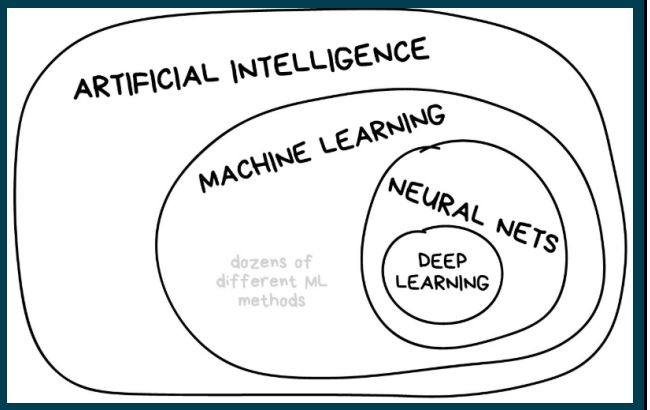


Example Machine Learning Workflow



What is Deep Learning?

- Deep learning is the use of multi-layered artificial neural networks
- An artificial neural network is a form of Machine Learning
  - Learns by training on many input-output pairs
  - Universal function approximator



Deep Learning

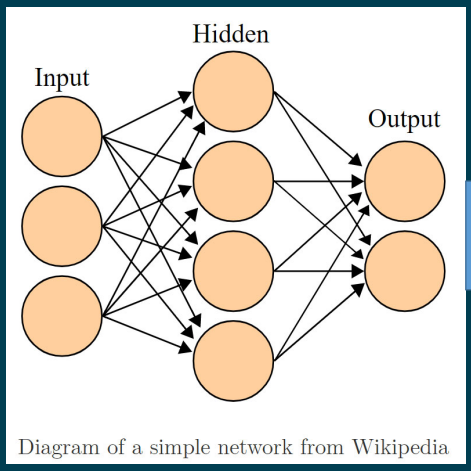
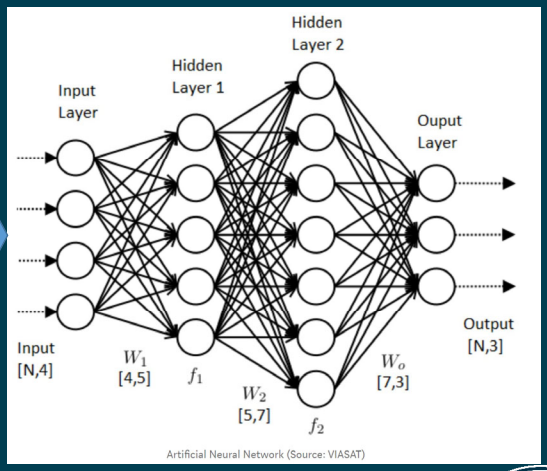


Diagram of a simple network from Wikipedia



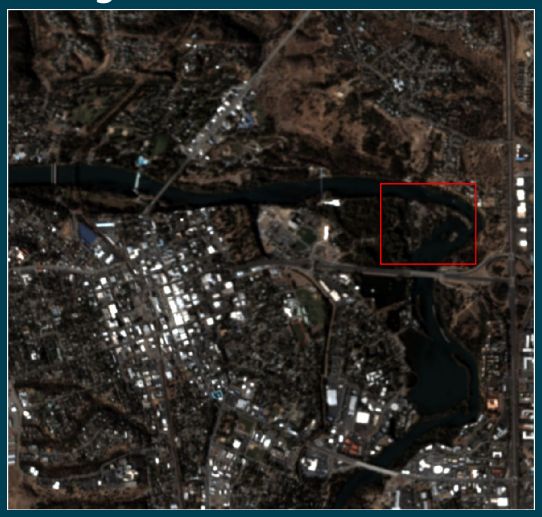
Artificial Neural Network (Source: VIASAT)

Conceptual Options for Delineation

- What if, given sufficient temporal and spatial resolution, satellite imagery were a time-series?
- Pixel or Object?
- Unsupervised or Supervised?
- Image, Numeric Time-Series, or Feature?

Satellite Imagery as a Time-Series (SITS)

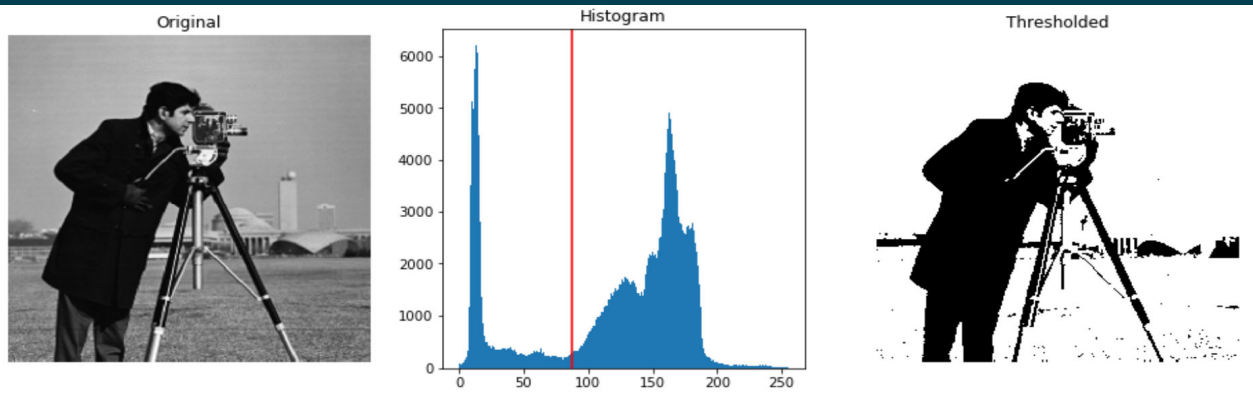
Redding from 10/11/2016 - Planet



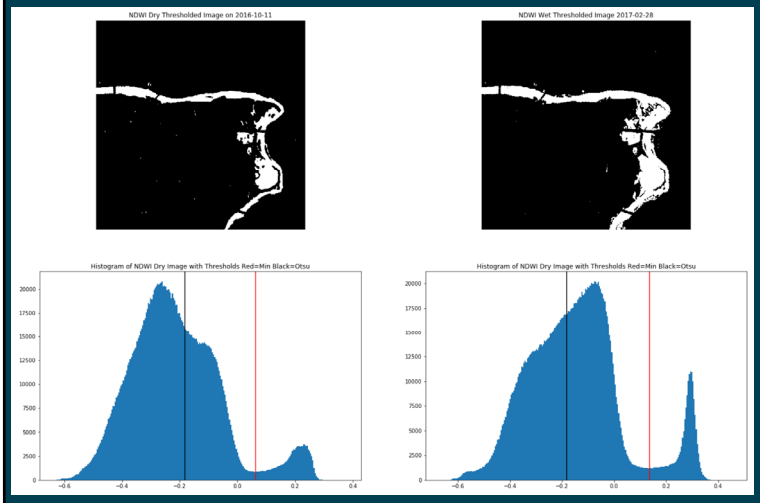
Redding from 02/28/2017 - Planet

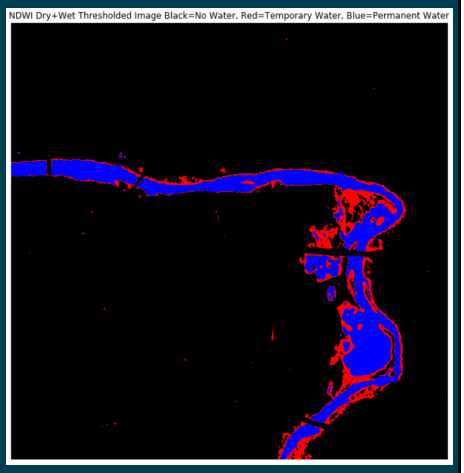
Classical Approach: Thresholding

- Image is converted to greyscale or greyscale-like
- Histogram of pixel values segmented at a threshold point
- Binary (0-1) segmentation of higher or lower than threshold point



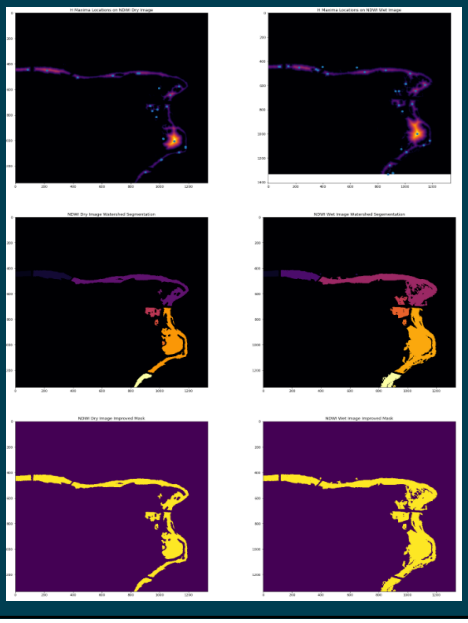
Traditional NDWI Thresholding



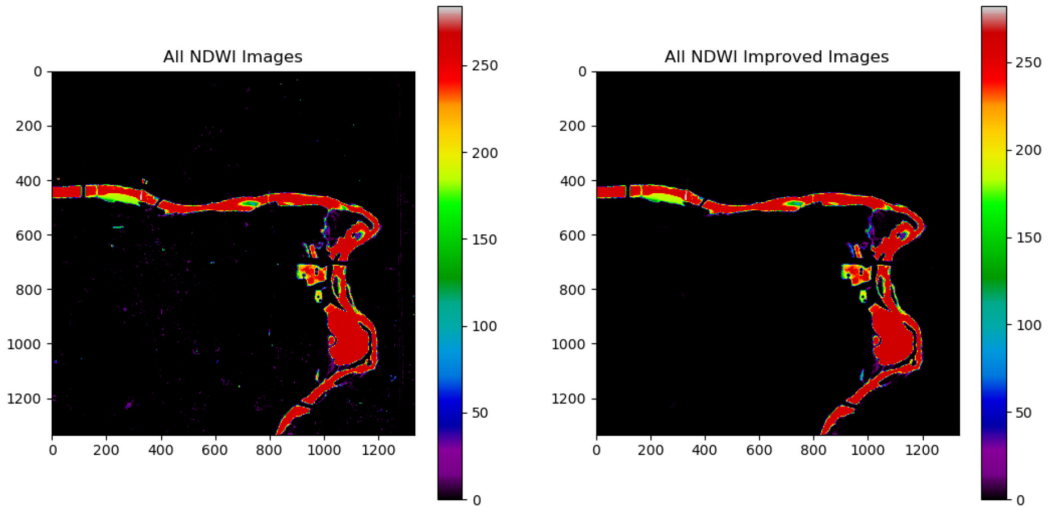


Enhanced NDWI Thresholding

- Over many images in an SITS NDWI stack detection error is additive.
  - Buildings
  - Roads
  - Bare Soil
- An enhanced NDWI thresholding method using local maxima and watershed segmentation algorithm reduces this error.



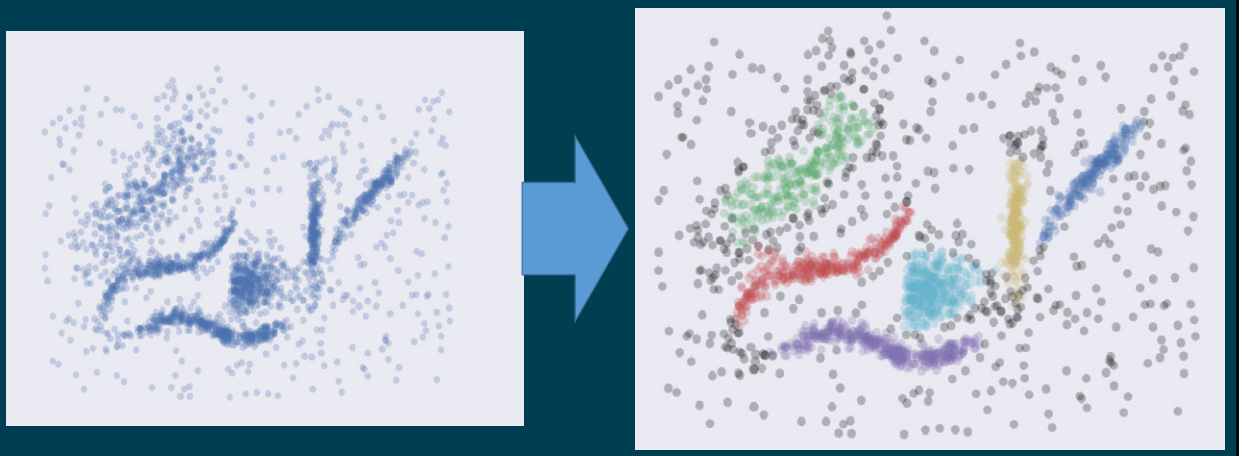
Enhanced NDWI Thresholding Result



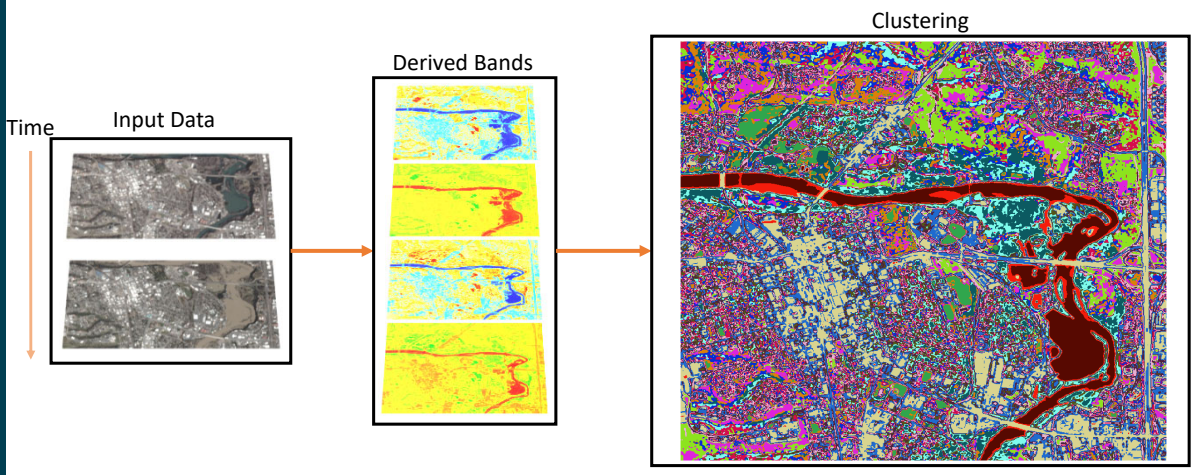
Thresholding Issues

- Each image in time must be looked at by a trained professional
- Two threshold values must be chosen
- Assumes bimodal distribution (land and water pixels)

Unsupervised Approach: Clustering



Experimental Clustering Approach

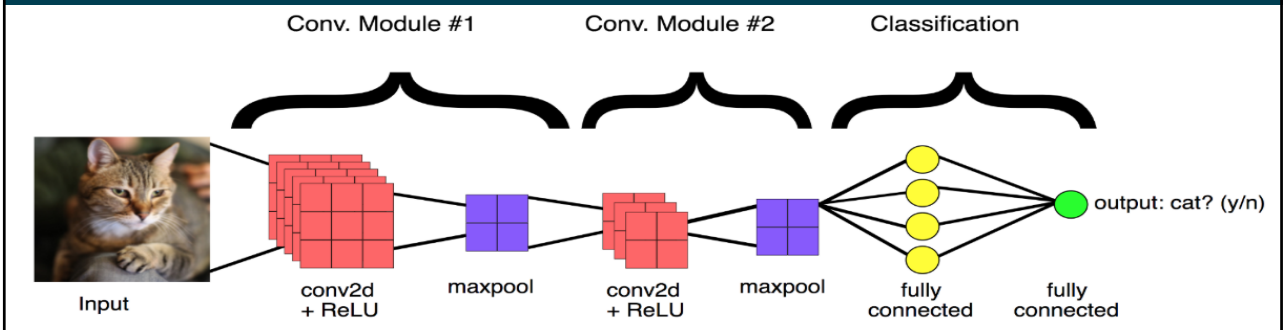


Clustering Issues

- Each cluster output image must be viewed to determine which "cluster" is temporary floodplain
- Clustering algorithm and hyperparameter choice
- Requires geospatial and machine learning professional
- Difficult to access cluster-quality

Supervised Approach: Deep Learning

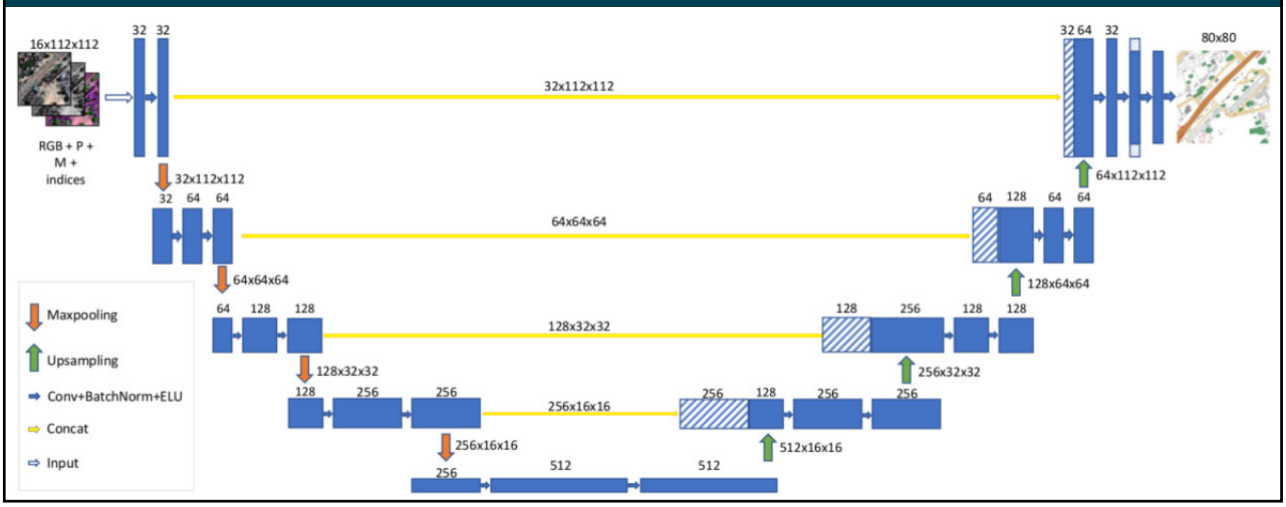
- Convolutional Neural Networks (CNNs) are a type of neural net
- CNNs are SOTA for image-based deep learning



Deep Semantic Segmentation

Deep Semantic Segmentation: U-net

Originally built for biomedical images and now being applied to satellite imagery



U-net Building Detection

Training Input and Label Pair

U-net Building Detection Prelim Results

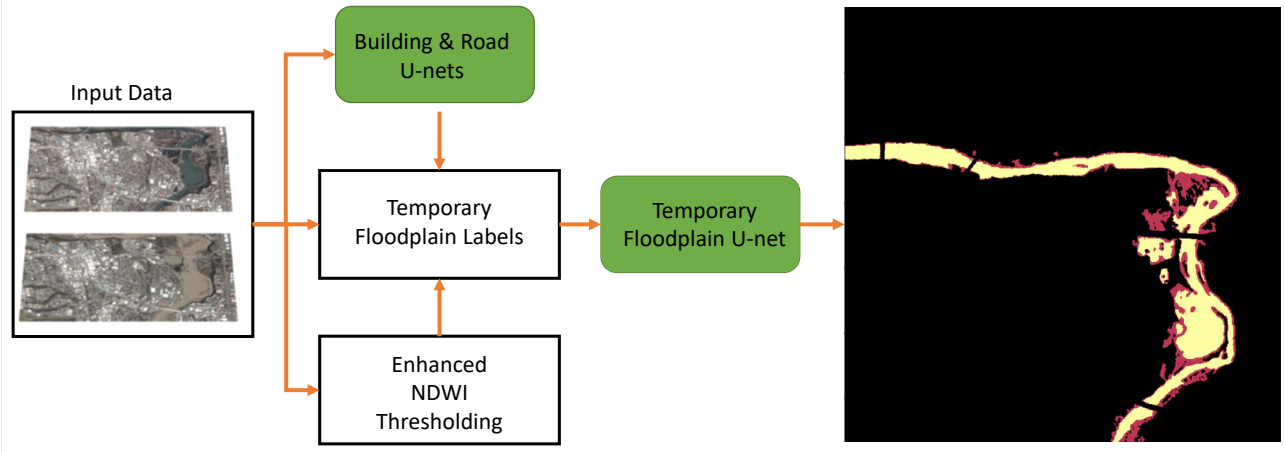
Training Phase 1

Training Phase 2

Why buildings when we want Temporary Floodplain?

- Temporary Floodplain labels are very difficult to come by
- Enhanced NDWI Thresholding can be used to create a set of "noisy labels"
- Unfortunately those "noisy labels" really like buildings, roads, and bare soil
- Using U-nets for buildings and roads we can mask out "noisy labels", improving the temporary floodplain labels
- The temporary floodplain U-net is the same as the other U-nets and form a U-net Ensemble

U-net Workflow in Progress



Next Steps: U-net Iteration

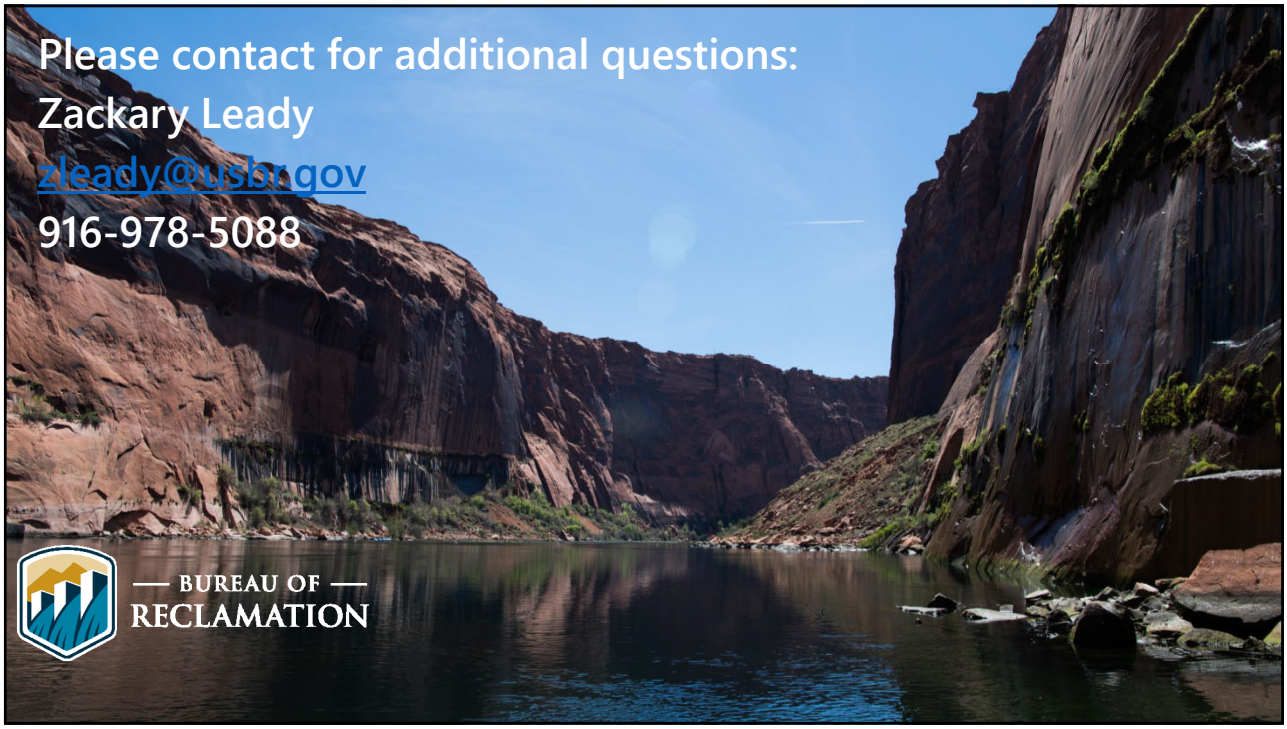
- Development of more robust training and test datasets
- Ensemble U-net
- Sentinel-2, SAR data fusion

Current Concerns

- Bare soil field confusion with flooded field
- AROSICS (open-source, python) co-registering accuracy
- Lack of Planet data moving forward, conversion to Sentinel-2
- Processing of 8 TB of data

Future Projects

- Hydrologic Models
  - Entity-Aware LSTMs
- Crop Classification (ET)
  - Input to ET models (water demand)
- Concrete crack detection
- Forecasting


Please contact for additional questions:
Zackary Leady
zleady@usbr.gov
916-978-5088
BUREAU OF
RECLAMATION