Part 2 - Technical Report

Table of Contents

Table of Figures

Table of Tables

1. Introduction

This document has been provided as supplementary support for the research paper. This technical report will provide additional details including further background readings, methodological details including feature selection and training and application of the classification models, and finally recommendations that future research may wish to consider.

2. Extended Literature Review

2.1 Existing Methods for Monitoring Plastic Pollution in the Mekong Region

Several studies in Southeast Asia have focused on identifying the sources and fate of macro plastics to refine waste management strategies (Rinasti et al., 2022; Tran-Thanh et al., 2022). Tran-Thanh (2022) demonstrated the potential of using multi-source geospatial data in GIS. This approach incorporates static indicators such as population density, economic status and topographical data with dynamic information such as local polices to model key hotspots of plastics pollution and identify plastic leaks into the Mekong River in Vientiane and Ubon Ratchathani. Despite the use of a wide range of indicators, the model was only marginally effective, correctly identifying plastic litter locations 29% of the time. Therefore, this highlights the challenge of using proxy indicators into a GIS that capture the sources, transport routes and accumulation sites comprehensively. Such models often miss the nuanced spatial variability of plastic waste that depends on established local knowledge of illegal dumping sites.

Given these limitations, direct observation through field survey is crucial for capturing the actual distribution of plastic waste. Many local initiatives for plastic monitoring in Vientiane focus on surveillance in the Mekong River, such as those employed by the MRC and UNEP (MRC, 2023; UNEP,2022). These efforts involve community-level sampling, visual observation, and use of smart technologies in the real-time detection of plastic waste. However, these initiative face challenges such as sporadic field survey collection times, requirement of extensive manpower and time-consuming processes. These challenges underscore the need for a more efficient methodology for plastic waste surveillance, rather than field surveys and modelled techniques alone.

2.2 PlanetScope Satellites

Recent developments in Earth Observation (EO) satellite constellations, such as PlanetScope, offer significant potential for mapping urban environments at high temporal resolutions (Valman et al., 2024). PlanetScope imagery provides relatively fine spatial resolution, capturing images at 3 to 5 meters, depending on the individual satellite's altitude (Planet Labs, 2022). Additionally, PlanetScope satellites have a high temporal resolution, with the capability to revisit the same location up to 12 times per day, which is crucial for monitoring dynamic environments and changes over time (Planet Labs, 2022). Karakus (2023) proposes that satellite products such as SkySat and PlanetScope, along with WorldView-3, will guide future research in monitoring marine debris and suspected plastics, addressing the low spatiotemporal resolution limitations of Sentinel-2. Among these, PlanetScope imagery from Planet Labs is the most accessible, available free of charge through the Planet Labs Education and Research Program.

Planet Labs' PlanetScope constellation includes different generations of satellites, each with distinct capabilities: Dove Classic, Dove-R, and SuperDove. SuperDove satellites offer 8 spectral bands, enabling more in-depth spectral analysis, which is crucial for identifying plastic characteristics from remote sensing platforms (Planet Labs, 2022). These bands are also interoperable with Sentinel-2 bands, which is significant as many spectral indices have been optimised to work with Sentinel-2 (Frazier and Hemingway, 2021).

These advancements make SuperDove satellites particularly suitable for applications requiring high precision, spectral diversity, and frequent temporal monitoring, such as detecting and monitoring plastic waste in urban settings.

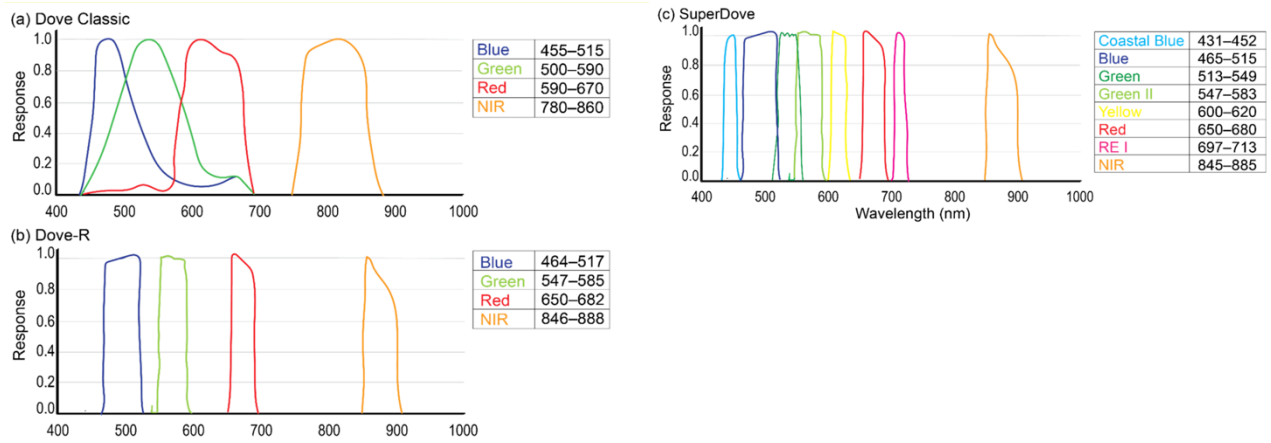


Figure 1.Comparison of PlanetScope Satellites

2.3 Comparison of Random Forest and Support Vector Machine

2.3.1 Random Forest Algorithm

Random Forest (RF) is an ensemble machine learning algorithm that constructs multiple decision trees and outputs the mode class or mean prediction of the individual trees (Figure 2). Developed by Breiman (2001), RF is particularly effective in handling complex classification tasks due to its ability to manage large datasets with high accuracy and resilience to overfitting (GeeksforGeeks, 2024). In the context of detecting plastic waste, RF models have been employed to analyse RGB, multispectral, and hyperspectral images to differentiate plastic materials from other land cover types (Chen 2017; Mifdal, 2021; Satki, 2023; Iordashe, 2022; Kalonde, 2022).

Feature selection is crucial in enhancing the performance of RF classifiers (Lary et al., 2016). While deep learning algorithms, such as Convolutional Neural Networks (CNNs), can automate feature selection by learning from the data, manually selecting features, as required for RF, can lead to better control and understanding of the classification process and is preferred when dealing with small training datasets, as in this study.

Additionally, altering the parameters of RF models can improve classification. This includes the number of trees generated (N-tree) and the number of features randomly selected at each split (M-try). Increasing M-try, whilst reduces computation speed, increases the correlation between trees and the strength of each individual tree, thereby impacting overall classification accuracy (Cortesi et al., 2021).

Two additional pieces of information from Random Forest algorithms can be determined: the measure of predictor feature importance and the proximity of input feature classes (Liaw and Wiener, 2002). These capabilities provide advantages over CNNs, particularly in terms of model interpretability.

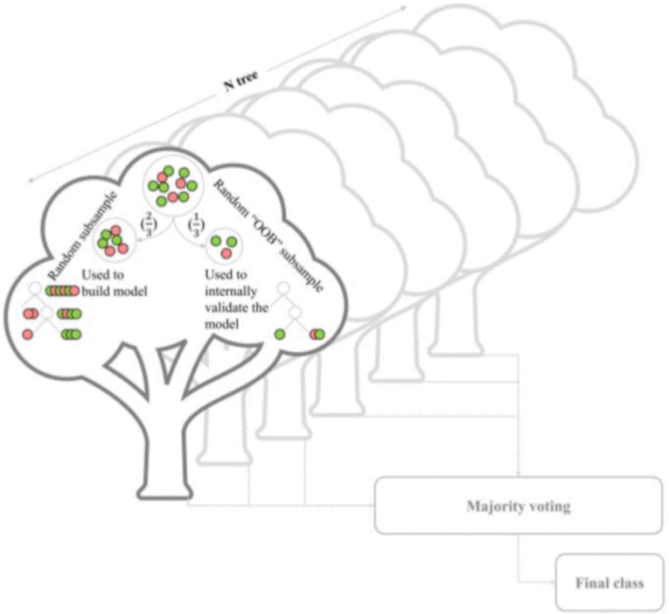


Figure 2. Random Forest Model Diagram

2.3.2 Support Vector Machine Algorithm

Support Vector Machines (SVMs) are a powerful tool for object detection, particularly effective for binary classification tasks. SVMs work by identifying a linear discriminant function that maximises the margin between different classes, creating the most discrete classes with the largest separation possible (Figure 3) (Cortes and Vapnik, 1995).

When data points cannot be separated linearly, nonlinear transformations can be applied using kernel tricks, mapping the data into a higher-dimensional space where a linear separation is not possible (Cortesi et al., 2021). The performance of SVM largely depends on the suitable selection of a kernel function (Sheykhmousa et al., 2020). Kernel functions include polynomial, radial base function, and sigmoid kernels.

In the context of object detection, SVMs have been widely used in applications from image classification to text categorisation. For example, SVMs have been successfully applied to hyperspectral image classification, where they effectively handle the complex and high-dimensional spectral data for niche land classes (Melgani & Bruzzone, 2004). Furthermore, SVMs are often integrated into more complex systems, such as combining with Histogram of Oriented Gradients (HOG) for pedestrian detection in images, demonstrating their versatility and effectiveness (Dalal & Triggs, 2005).

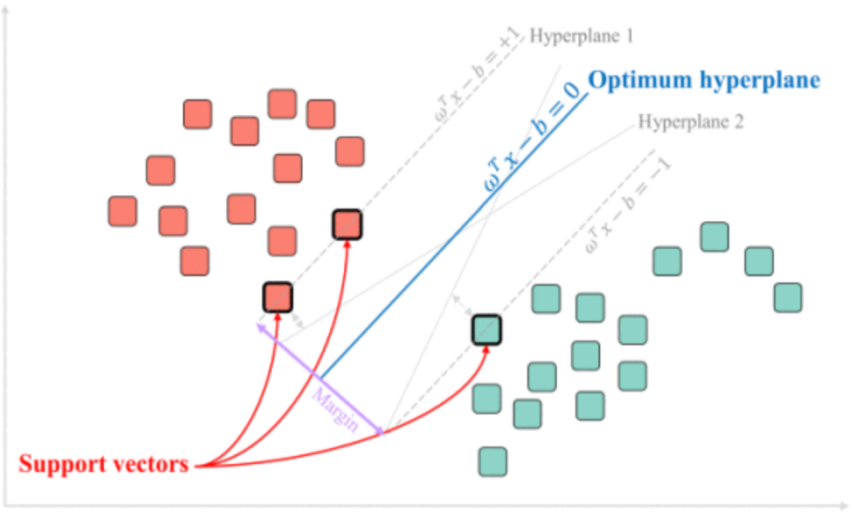


Figure 3.Support Vector Machine Diagram

2.3.3 Random Forest vs Support Vector Machine for Remote Sensing Image Classification

RF and SVM are both well-known, top-ranked machine learning algorithms (Sheykhmousa et al., 2020). Medium spatial resolution imagery is most commonly used for SVM, while high spatial resolution imagery is often preferred for RF. There is no strong correlation between the acquired accuracies and the number of features for SVM and RF methods. However, Sheykhmousa suggests that SVM may be more accurate when classifying data with many features compared to RF. In UAV-based land cover classification studies, both algorithms have shown high overall accuracy. For instance, RF and SVM both produced high classification accuracy for urban vegetation mapping using UAV data, with RF being particularly effective in integrating texture analysis (Feng et al., 2015).

3. Data Sources

3.1 Vientiane Field Survey 2024

3.1.1 Planning the Survey

The survey was initially organised to provide potential ground truth data for PlanetScope and UAV imagery and to gain insights into the dimensions and characteristics of plastic waste across Vientiane. As the UAV imagery later aligned with the pLitter survey, this survey supported the latter purpose. Additionally, the data collected will be made public to support future research on plastic waste in Vientiane.

The field survey was conducted on June 29-30th, 2024. It was led by advocacy group founder Serge Doussante, with assistance from staff at the National University of Laos.

The survey aimed to collect data from 100 sites across the Ban Hof agricultural region. Sites were selected based on anticipated plastic accumulation areas, such as alongside main roads and open grounds (Figure 4). This selection was informed by local knowledge from supervisor Peter-John Meynell and Serge Doussante.

At each site, surveyors searched within a 20m radius of the coordinates to locate and document plastic waste.

A preparatory document was provided to guide the survey, including:

1. Instructions for accessing a .kml file with the field survey site locations.
2. Guidelines for photographing plastic waste, including both bird's-eye and contextual views.
3. Notes to record: date-time, plastic dimensions (m), and type of plastic waste.
4. Instructions for submitting information, ensuring all photos were geotagged.

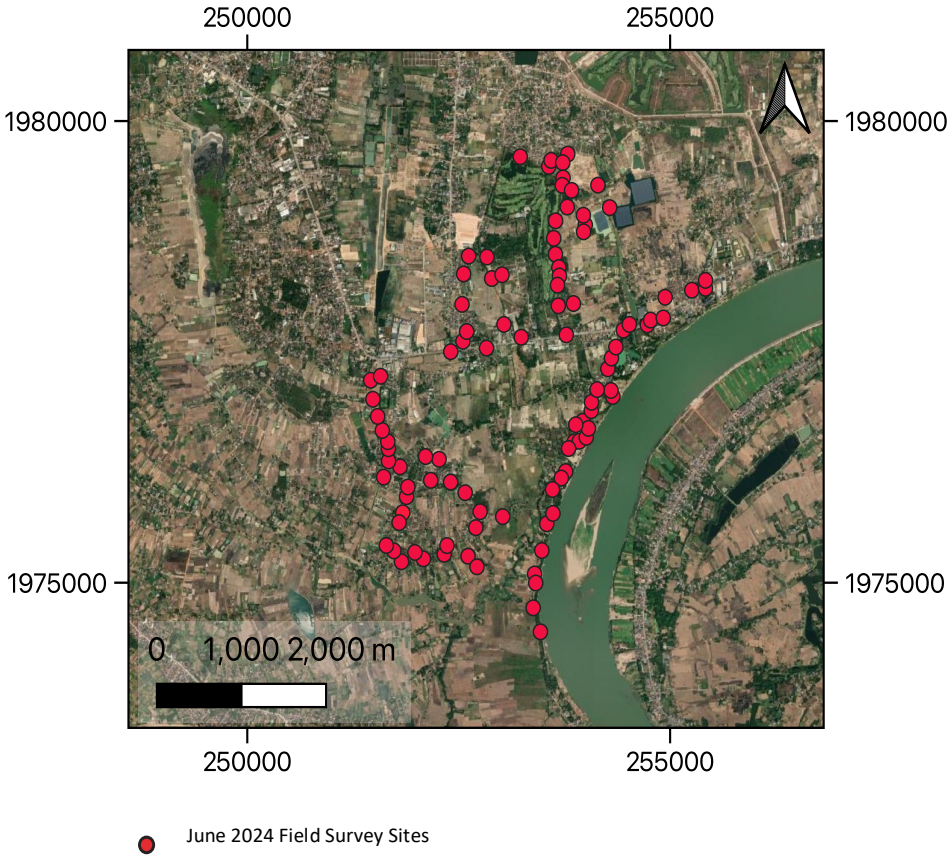


Figure 4. Plastic Survey Field Sites, June 2024

Figure 5. Photos from Plastic Survey 2024

3.1.2 Field Survey Results

The survey results indicate that single-use plastic bags are the most prevalent form of plastic waste, followed by plastic bottles and Styrofoam (Figure 6). Additionally, an analysis of photographs reveals a significant presence of litter from food packets, such as crisp packets.

Two critical observations emerged from the survey:

1. Mixed Waste Context: Plastic waste is frequently found intermixed with other types of waste, including cardboard, paper, cans, glass bottles, and wood planks (Figure 5a). This intermixing complicates the characterisation of plastic waste, highlighting the challenge of distinguishing plastic from other waste types present.

2. Household Waste Disposal: Plastic bags filled with household waste are often observed on the streets in residential area, awaiting collection by waste management services (Figure 5d). It is essential to distinguish this planned legal disposal from the mapping of plastic waste intended for study, as the focus should be on unregulated plastic waste.

In terms of plastic waste dimensions, the median height was determined to be 0.3m, with a median area of 3.75m^2 (Figure 6). The plastic waste documented primarily consists of numerous sporadic smaller items, with a few much larger accumulations. These larger clusters are likely indicative of common areas for repetitive illegal dumping of plastic waste.

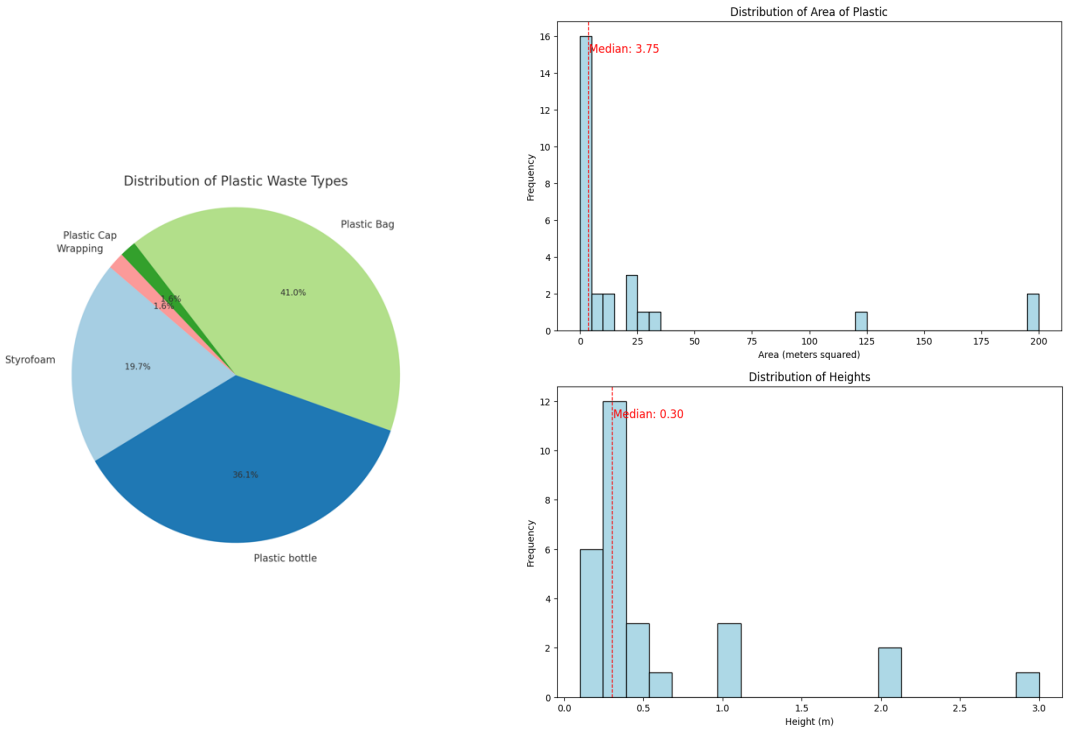


Figure 6. Plastic characteristics - type and dimensions

3.2 Existing Data Sources

3.2.1 Downloading Planet Imagery

The following steps were undertaken to access PlanetScope imagery. First, the Planet Explorer website was opened in a web browser (https://www.planet.com/explorer/). To define a specific area of interest, a KML file used for defining drone imagery regions was imported. The date range was set to correspond with the collection dates of the pLitter data in the region, specifically from November 20th to December 8th. All available imagery within this period was manually reviewed. Surface reflectance corrected SuperDove imagery from November 23rd was then selected for download. Although the images were filtered to 0% cloud cover, some variation in image quality was observed.

3.2.2 Accessing PLitter Data

The PLitter data is available on ArcGIS Online: (https://www.arcgis.com/home/item.html?id=18c67da0e9fe401ebdd8cd2cd6c0ea8b). The PLitter initiative was conducted across several cities and countries, primarily in Southeast Asia. Hence, after downloading the data into QGIS, the plastic survey points were subset to focus on Vientiane.

4. Data Analysis Preparation

4.1 Computational requirements

4.1.1 Software

Table 1. Software Requirements

Software	Version	Uses
Excel	16.63.1	Storing information such as field survey data, training data features, model predictions, accuracy results
QGIS	3.36.3 Maidenhead	Displaying Planet and UAV imagery and field survey points, creating training data, creating validation data, displaying model outputs
Visual Studio Code	1.91.1	Accessing Python through the VSC extension; providing a good user interface.
Python	v2024.10.0	Calculating features from training data, developing SVM and RF models, preparing drone imagery for model application, applying models. Also used to create graphs such as box plots, pie charts and bar charts.

4.1.2. Hardware

The majority of processes were run on a personal laptop, while the GEOS workstation was used to accelerate specific tasks.

Table 2.Hardware Requirements

Device	Name	Processor	Memory	Operating System	Primary Usage
Personal Laptop	N/A	MacBook Pro, Apple M2 chip	8 GB	macOS (ARM64)	- Created training data - Developed models - Prepared drone imagery for model application - Applied models
GEOS Workstation	geos-w-069	12th Gen Intel® Core™ i9-12900 @ 2.40 GHz	192 GB RAM	64-bit Operating System	- Tiling drone imagery - Applying models - Mosaicking tiles back together to create final prediction maps while running additional tasks simultaneously

4.1.3 Python Packages

Table 3. Python packages and versions used

Application	Libraries Used (Version)
Features Code	os (Standard Library), pandas (2.1.2), gdal/osgeo (3.9.0), skimage (0.23.2), scipy (1.13.1), opencv-python (4.10.0.84)
Model Code	os (Standard Library), pandas (2.1.2), matplotlib (3.8.1), joblib (1.4.2), plotnine (0.13.6), scikit-learn (1.5.0), openpyxl (3.1.5)
Applying Model	os (Standard Library), pandas (2.1.2), numpy (1.26.1), skimage (0.23.2), rasterio (1.3.10), gc (Standard Library), joblib (1.4.2)

4.2 Training Data Generation

4.2.1. Initial Training Data

To train a machine learning model for plastic classification, diverse training data representing different land classes was essential. The selected classes included plastic, vegetation, water, earth, roads, and buildings, identified by examining the available drone imagery.

Initially, the training data consisted of subsets of the drone imagery in the form of pixels from irregular polygons (Figure 7). In QGIS, polygons were drawn over regions for each land class, and then these pixels were extracted. For the plastic training data, relevant drone imagery was identified by overlaying a the pLitter plastic survey data from 2021. All training data was created in the Thatluang Economic Lake Region, as the greatest ground survey points were located in this region.

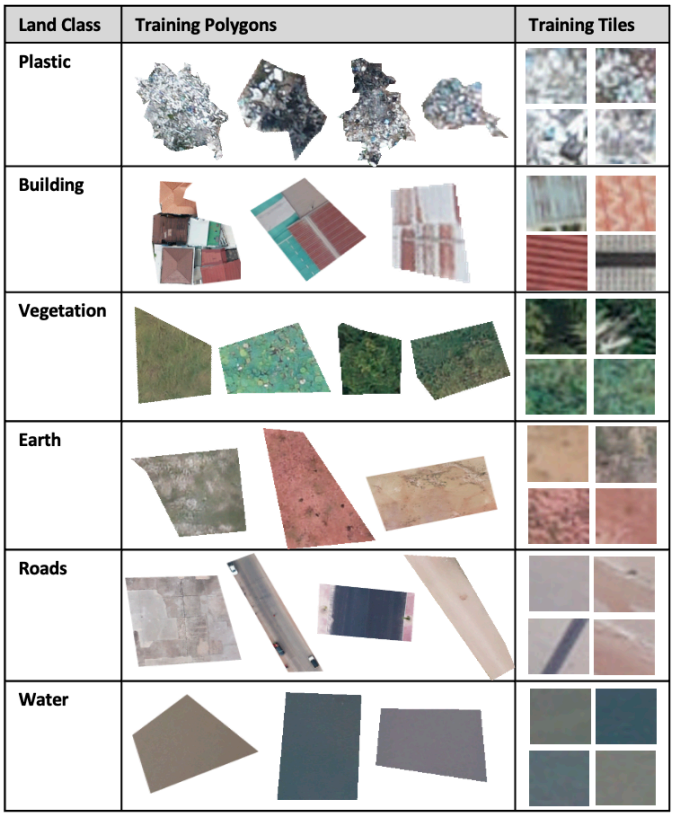


Figure 7.Examples of training polygons and tiles extracted from the drone imagery

Plastic waste areas were delineated based on the locational of photos as well as coordinates, with the photos providing context for the accurate creation of polygons around plastic waste piles in the imagery (Figure 8). Sixty polygons were created in total.

For the other land classes, relevant drone imagery to extract was based on human interpretation. Sixty polygons were created for each land class. All training polygons were then merged, retaining their original layer paths as attributes. The drone imagery for region was clipped to each polygon, generating 360 raster images stored as .tif files. An alpha band was generated to ensure that a black background did not influence the feature characteristics of the training images.

Figure 8.Example usage of pLitter survey data when determining plastic training data

4.2.2 Improved Training Data

Following an analysis of the initial model results, the training data was refined to better align with the images that the model would eventually analyse during its application. The irregular polygons were converted into regular square tiles of 28x28 pixels, resulting in the generation of thousands of training data tiles. Given the large number of tiles generated, 120 representative images were randomly selected for each land class. Although a larger training dataset would be ideal, the training data size was reduced due to the constraints of computational resources. Additionally, many tiles did not contain a complete tile of the RGB data necessary, due to the irregular shapes of the original polygons. These incomplete tiles were manually removed, resulting in a refined set of training data.

5. Developing Feature Characteristics

Feature Type	Feature Name	Description
Optical Properties	RGB Mean	Mean intensity of the RGB channels
	RGB Std	Standard deviation of the RGB channels
	HSV Mean	Mean values of Hue, Saturation, and Value
	HSV Std	Standard deviation of Hue, Saturation, and Value
Haralick Texture Features	Contrast	Measure of local intensity variation
	Dissimilarity	Measure of how different the elements are
	Homogeneity	Measure of closeness of the distribution of elements in the GLCM to the GLCM diagonal
	ASM (Angular Second Moment)	Measure of textural uniformity
	Energy	Sum of squared elements in the GLCM
	Correlation	Measure of how correlated a pixel is to its neighbour over the whole image
	Entropy	Measure of randomness
	Inverse Difference Moment	Measure of image homogeneity
	Inertia	Measure of the intensity contrast between a pixel and its neighbour over the whole image
	Cluster Shade	Measure of the skewness and uniformity of the GLCM
	Cluster Prominence	Measure of the kurtosis of the GLCM
	Haralick Correlation	Measure of the linear dependency of grey levels on those of neighbouring pixels

Table 4. Explanation of Textural and Optical Features

As with many machine learning models such as RF and SVM, image statistics or features are often manually defined as inputs to the model (Lary et al., 2016). This section explains which features were selected and why they were chosen to distinguish plastic from other land classes. Two kinds of features were chosen: optical and textural. Table 4 lists all the features identified and evaluated for use in the classification models.

5.1 Optical Characteristics in Image Classification

RGB (Red, Green, Blue) and HSV (Hue, Saturation, Value) colour spaces are critical in image classification tasks due to their ability to capture and differentiate various land features based on colour and intensity. The mean values of RGB and HSV channels provide summary statistics that determine the average colour and brightness of an image or region.

The standard deviation measures the variability or spread of colour and brightness values. A low standard deviation indicates that the colours are similar and consistent, while a high standard deviation suggests a wide range of colours and intensities. This measure is particularly useful for identifying heterogeneous regions such as plastic waste in mixed environments.

5.2 Textural Characteristics in Image Classification.

Textural features are a crucial aspect of image analysis and classification, enabling the differentiation of various regions in an image based on their textural properties. These features are particularly valuable in land classification for remote sensing applications (Chen et al., 2016), facilitating the detection of specific materials from an aerial perspective, such as plastic waste. Plastic waste exhibits distinctive textural characteristics due to its spatially heterogeneous nature, making it identifiable in aerial images. Consequently, Haralick texture features were computed and used as inputs for the RF and SVM land classification models to enhance the accuracy of detecting plastic waste in UAV imagery.

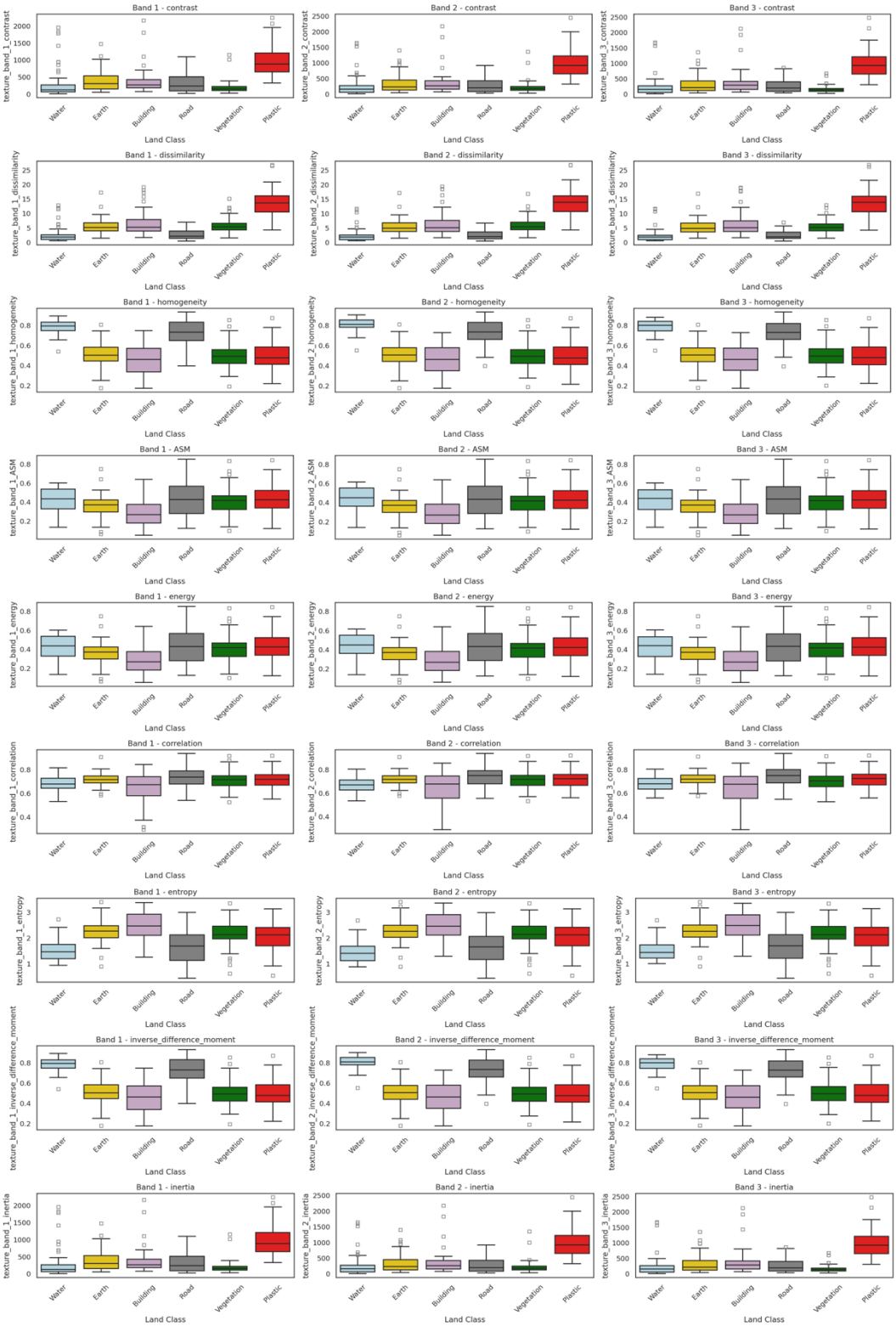
The Gray-Level Co-occurrence Matrix (GLCM) average proposed by Haralick in 1973 (Haralick et al., 1973) is one of the most widely accepted and utilised statistical methods to determine texture characteristics (Chen et al., 2016). The GLCM is a function of the direction and distance relationship between two neighbouring pixels, recording the number of co-occurrence relationships between one pixel and its specified neighbouring pixels for a specific direction (angle) and distance.

This paper calculated the GLCM in one direction ($\theta = 0°$) and over a distance of 1 pixel. By focusing on one distance and direction, this study explored and analysed a broad range of texture features and reduced the computational strain, making the process more efficient and feasible on the available hardware. Future studies may wish to consider additional directions and distances.

Initially, texture features were calculated across each RGB band, creating a GLCM for each. However, the features displayed very similar characteristics across the three bands; hence,

these were averaged moving forward. This approach also reduced computational requirements.

To further understand the distribution and variability of these texture characteristics across all land classes, box plots were generated (Figure 9). After analysing these box plots, a set number of features were selected for model input: average contrast, dissimilarity, homogeneity, entropy, inverse difference moment, cluster shade, and cluster prominence. The chosen features exhibited distinct distribution patterns for the different land classes and showed relatively low correlation with each other, ensuring a comprehensive and diverse set of inputs for the classification models.


Band 1 - contrast
Band 2 - contrast
Band 3 - contrast
Band 1 - dissimilarity
Band 2 - dissimilarity
Band 3 - dissimilarity
Band 1 - homogeneity
Band 2 - homogeneity
Band 3 - homogeneity
Band 1 - ASM
Band 2 - ASM
Band 3 - ASM
Band 1 - energy
Band 2 - energy
Band 3 - energy
Band 1 - correlation
Band 2 - correlation
Band 3 - correlation
Band 1 - entropy
Band 2 - entropy
Band 3 - entropy
Band 1 - inverse_difference_moment
Band 2 - inverse_difference_moment
Band 3 - inverse_difference_moment
Band 1 - inertia
Band 2 - inertia
Band 3 - inertia
Land Class
Water
Earth
Building
Road
Vegetation
Plastic

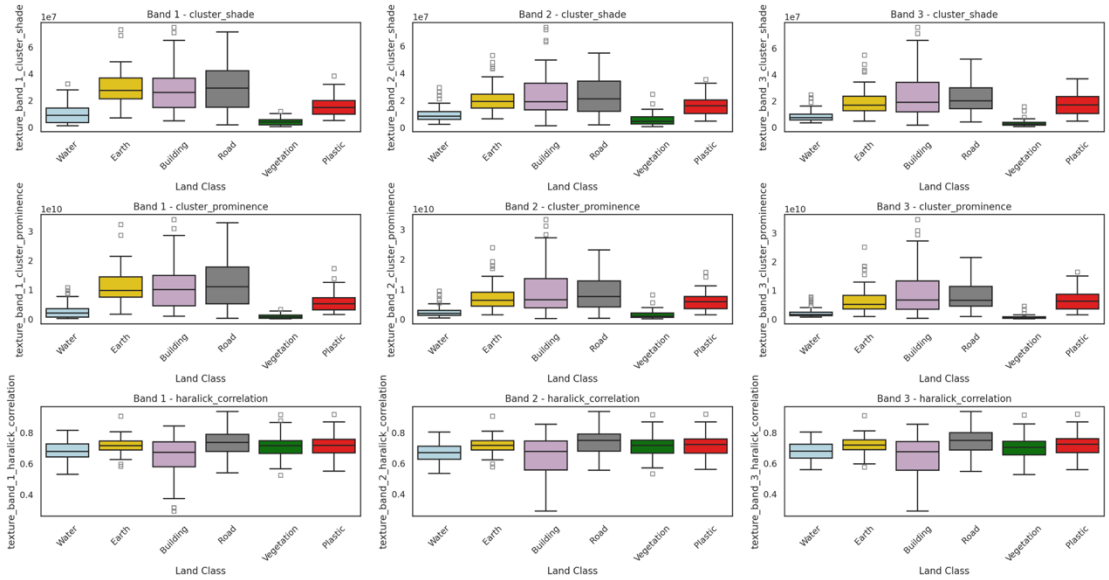


Figure 9. Boxplots showing distribution of feature characteristics for the training data.

6. Developing the Random Forest and Support Vector Machine Algorithms

In this section, the development and implementation of the RF and SVM algorithms for land classification is explored. Both the algorithms were developed using Python 3, employing the Random Forest Classifier and C-Support Vector Classification (SVC) model from the Scikit-Learn machine learning library. The number of features input into the SVM and RF models differed, with fewer features used in the SVM model due to its higher computational intensity (Sheykhmousa et al., 2020). Model parameters were fine-tuned to provide optimal models using a grid search approach.

6.1 Random Forest

The performance of the Random Forest (RF) model was evaluated by varying the number of trees (n_estimators) and the number of features (max_features). As shown in Figure 10, the model's overall accuracy ranged from 0.81 to 0.93 with varying parameters. Optimal accuracy was achieved with 12 features and 30 trees, and these parameters were subsequently used. During the development of the RF model, two additional calculations were assessed: feature importance and class proximity.

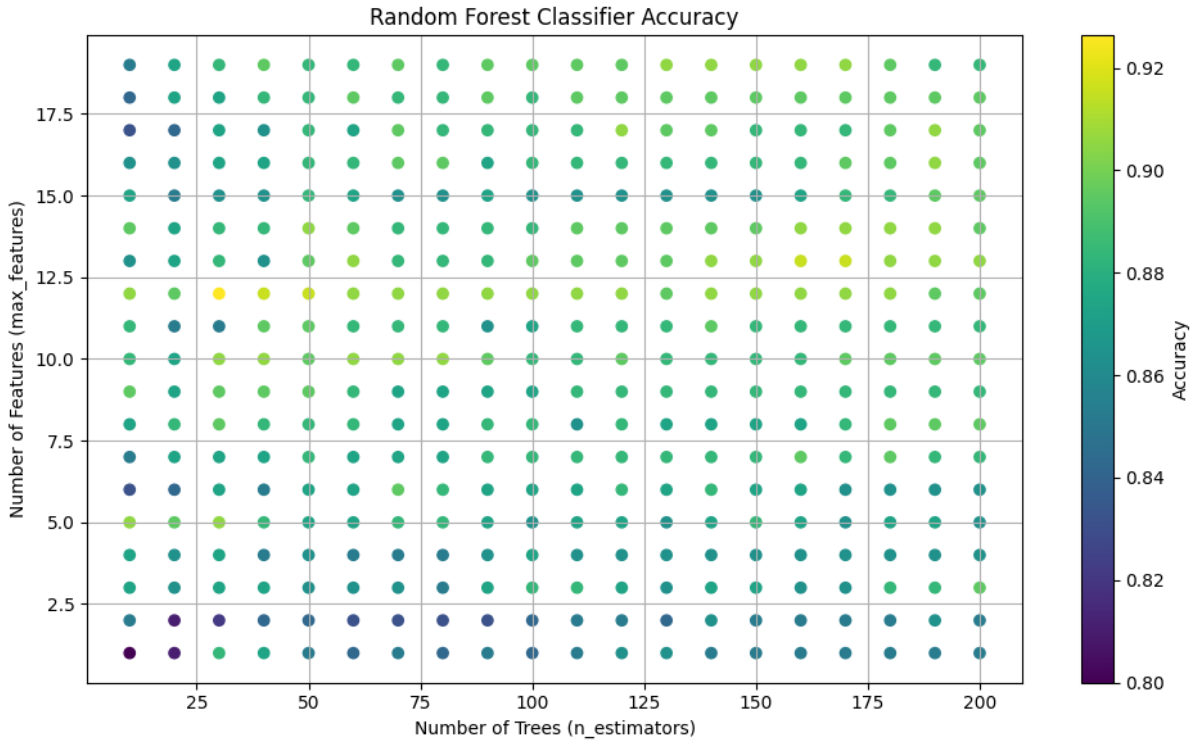


Figure 10.Cartesian Grid Search for optimal parameters for RF model

6.2 Support Vector Machine

For the SVM model, the features utilised were determined from the feature importance calculation for assessing the plastic class, derived from the Random Forest classifier. See Figure 12a for the 10 features used.

The model parameters were fine-tuned by searching for the best combination of the Regularisation Parameter (C), Kernel Coefficient (gamma), and kernel type. As shown in Figure 11, the model's overall accuracy ranged from 0.19 to 0.89, with the kernel type (rbf or linear) being the most significant factor. The best combination of parameters used for the SVM model was C=1, gamma=scale, and kernel=linear.

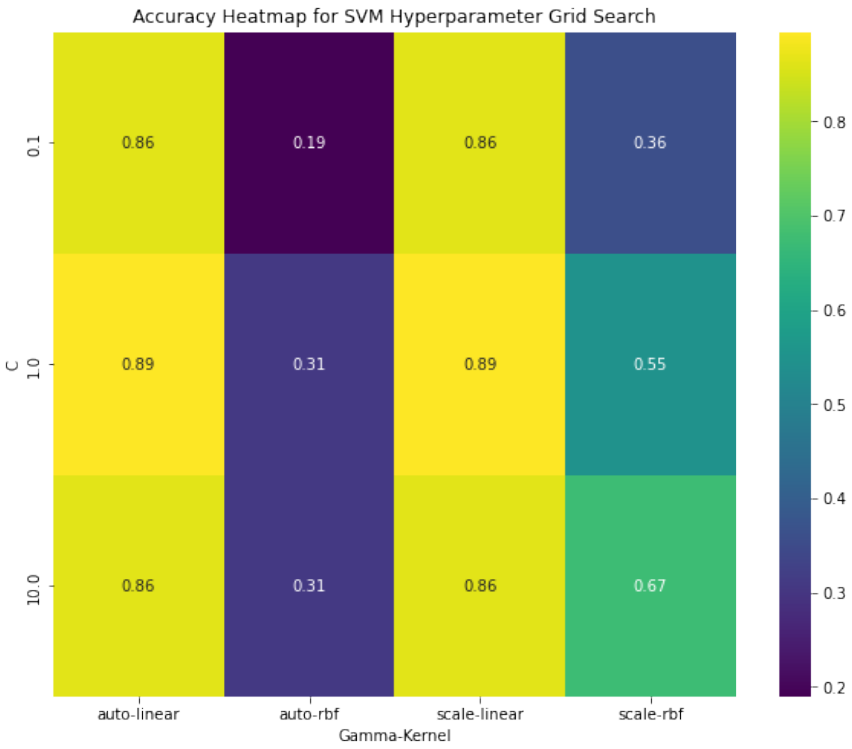


Figure 11.Cartesian Grid Search for optimal parameters for SVM model

6.3 Additional Metrics Calculated from the RF Algorithm

6.3.1 Feature Importance

The feature importance scores from the Random Forest Classifier indicate the significance of each feature in the prediction process. This study aimed to understand the critical features for classifying plastic. Hence, the results in the research paper analyse Figure 13a, which was created for a binary Random Forest Classifier of plastic versus non-plastic. However, to predict plastic in practice, the models predicted multiple land classes, one of them being plastic. Therefore, there are some discrepancies in feature importance when differentiating between six land classes rather than just plastic (Figure 12a versus 12b).

Both feature importance calculations indicate that 'mean saturation' and standard deviation values for RGB properties are important. It is noteworthy that texture features are less important when differentiating between various land classes compared to distinguishing plastic from non-plastic. This suggests that textural features are more significant characteristics for plastic than for other land classes.

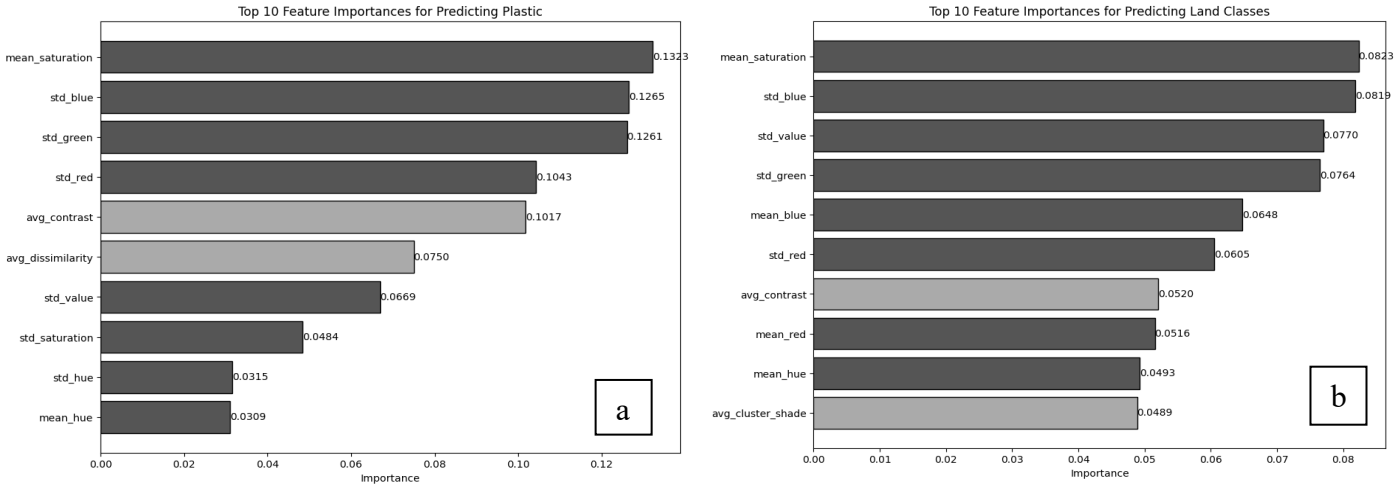


Figure 12.Top 10 important features for [a] Plastic versus Non-Plastic [b]Land Classes

6.3.2 Land Class Proximity analysis

For additional analysis, the proximity matrix was evaluated to understand the similarity between different land classes (Figure 13) (Liaw and Wiener, 2002). Higher proximity values represent similar features within or between datasets for different land classes, for example the water-water class value indicates minimal variation within the training data for this class. Ideally, different classes would have very low proximity values between them, indicating clear distinction between classes. When analysing the 'Plastic' class, the high proximity value of plastic-plastic indicates little variation within the training data. However, there is moderate proximity between plastic-road and plastic-building, suggesting the model may experience difficulty when classifying these land types due to overlapping features.

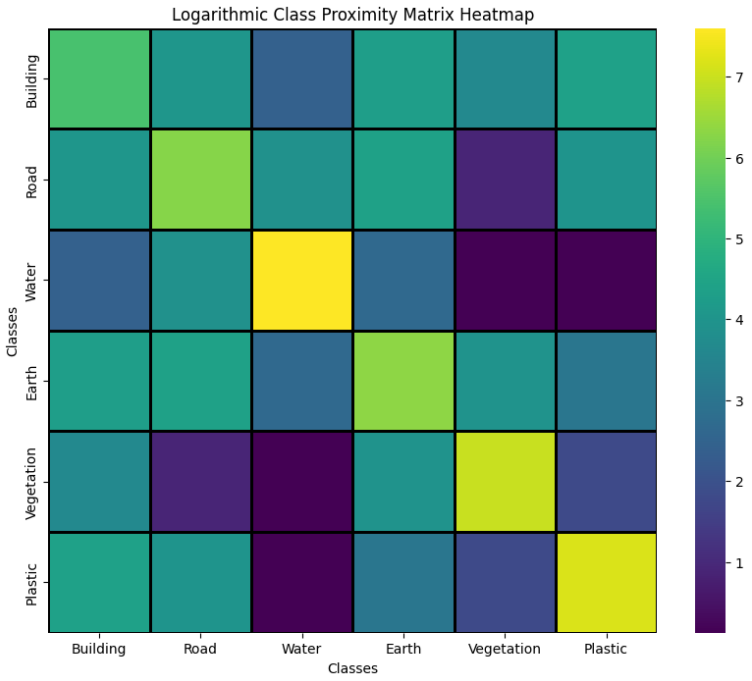


Figure 13.Class Proximity Matrix

7. Generating Model Accuracy Results

To evaluate the performance of the classification models, several statistical metrics were calculated. These metrics provide insights into the accuracy and reliability of the model predictions. The key statistics used to measure model accuracy are summarised in Table 5.

Metric	Definition	Formula
True Positive (TP)	The number of correct positive predictions.	*TP*
False Positive (FP)	The number of incorrect positive predictions.	*FP*
True Negative (TN)	The number of correct negative predictions.	*TN*
False Negative (FN)	The number of incorrect negative predictions.	*FN*
Precision	The ratio of correctly predicted positive observations to the total predicted positives.	$\frac{TP}{(TP + FP)}$
Recall	The ratio of correctly predicted positive observations to all observations in the actual class.	$\frac{TP}{(TP + FN)}$
F1 Score	The weighted average of Precision and Recall.	$\frac{2 * (Precision * Recall)}{(Precision + Recall)}$
Overall Accuracy (OA or Po)	The proportion of correctly classified instances out of the total instances.	*(TP + TN) / (TP + TN + FP + FN)*
Kappa Coefficient	A measure of agreement between predicted and actual classifications, correcting for chance.	$\frac{Po-Pe}{1-Pe}$

Table 5. Definitions and formulae of accuracy metrics

Accuracy results were generated twice, firstly during the initial training of the model and secondly during the application of the model. For the application accuracy statistics, 1000 validation points were generated randomly across a small segment of the drone imagery which is referred to as the 'validation site'. A manual land class prediction was then given to each of these points. Figure 14 displays the validation site and the validation points. The distribution of land class observations across the validation points is indicated in Table 6.

Land Class	Building	Earth	Plastic	Road	Vegetation	Water
Number of points	31	242	26	82	587	32

Table 6. Number of accuracy points generated for each land class.

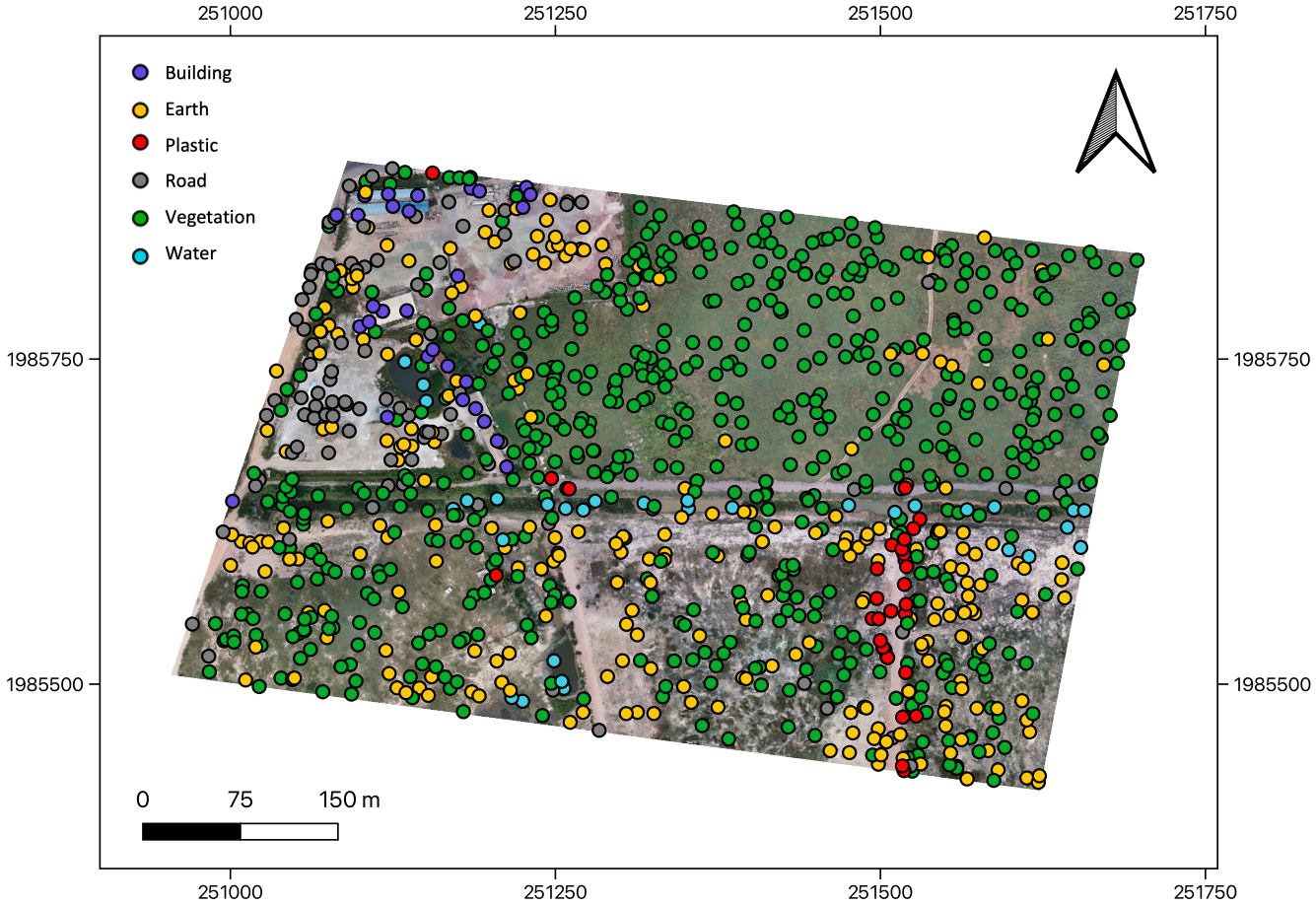


Figure 14.Validation Site with 1000 accuracy points of manual predictions.

8. Application of Model

This section explores the methods used to apply the developed models back to the drone imagery. Two primary approaches were investigated: first, tiling the drone imagery and applying a prediction to each square; second, utilising a segmentation approach where the drone imagery was divided into polygons based on spectral characteristics. Following segmentation, the model was applied to these polygons to classify the land features accurately. The best approach was used to apply the models in the Research Paper.

8.1 Tiled Approach for Model Application

To effectively apply the machine learning algorithms to the drone imagery, a tiling approach was employed. The large images were divided into smaller tiles of 28x28 pixels and 14x14 pixels, representing 1.4m² and 0.7m² on the ground, respectively. These tile sizes were chosen based on available field survey data, which indicated that the median plastic area was 3.75m². By using tiles under half this size, it is more likely to capture tiles that can be distinctly identified as plastics. This approach prevents larger tiles from including multiple land classes, which would complicate the predictions of the RF and SVM models.

To generate accuracy statistics for informed decision-making, the models were applied to a small segment of the drone imagery and compared with 1000 manually generated validation points (see Figure 14). The results can be seen in Table 7.

Model	Building	Earth	Plastic	Road	Vegetation	Water
14_pix_RF	0.25	0.46	0.36	0.07	0.17	0.79
14_pix_SVM	0.75	0.00	0.07	0.07	0.40	0.84
28_pix_RF	0.44	0.44	0.00	0.00	0.12	0.37
28_pix_SVM	0.56	0.02	0.00	0.03	0.32	0.84

Table 7. F1 scores for RF and SVM models applied to tiled drone imagery of 28x28 and 14x14 dimension.

It is important to note that these results are based on the initial set of training data (see section 4.2) and are not the same as the final accuracy results presented in the research paper.

These results show averaged precision and recall for each class for the RF and SVM models applied on 28x28 pixel and 14x14 pixel tiles. The findings indicate that the 14x14 pixel tiles yielded significantly higher accuracy, particularly for the plastic class. No plastic was accurately predicted using the 28x28 pixel tiles. These preliminary results informed the model application in the research paper.

8.2 Segmentation Approach for Model Application

Segmentation on a specific section of drone imagery was performed to explore another approach for applying machine learning algorithms. The 'validation site' was used to explore this approach. Using the open-source software Orfeo Toolbox through QGIS, the mean-shift algorithm was employed. This algorithm groups together pixels with similar spectral characteristics based on user-specified parameters.

The sensitivity of the mean-shift algorithm is controlled by the spatial radius and range radius parameters. The spatial radius defines the neighbourhood of pixels considered during processing, while the range radius sets a threshold for grouping pixels with similar values. A larger spatial radius results in more smoothing as it encompasses a larger area, and a greater range radius allows more variation in colour intensity within the grouped pixels (Orfeo Toolbox, 2024).

In this study, a spatial radius and range radius of 25 was utilised, following the work of Kalonde (2022), resulting in the segmented output shown in Figure 15. Predictions were applied to each segment or polygon, and the results were uploaded to QGIS for visual assessment. An accuracy assessment was carried out, with the results presented in Table 8.

Model	Building	Earth	Plastic	Road	Vegetation	Water
Segmentation RF (Precision | Recall)	0.04 0.45	0.67 0.02	0.13 0.76	0.36 0.71	0.81 0.40	0.33 0.06
Segmentation SVM (Precision | Recall)	0.17 0.5	0.46 0.57	0.06 0.17	0.00 0.00	0.76 0.43	0.00 0.00

Table 8.Precision and recall scores for the RF and SVM models when applied to a segmented drone imagery rather than tiled

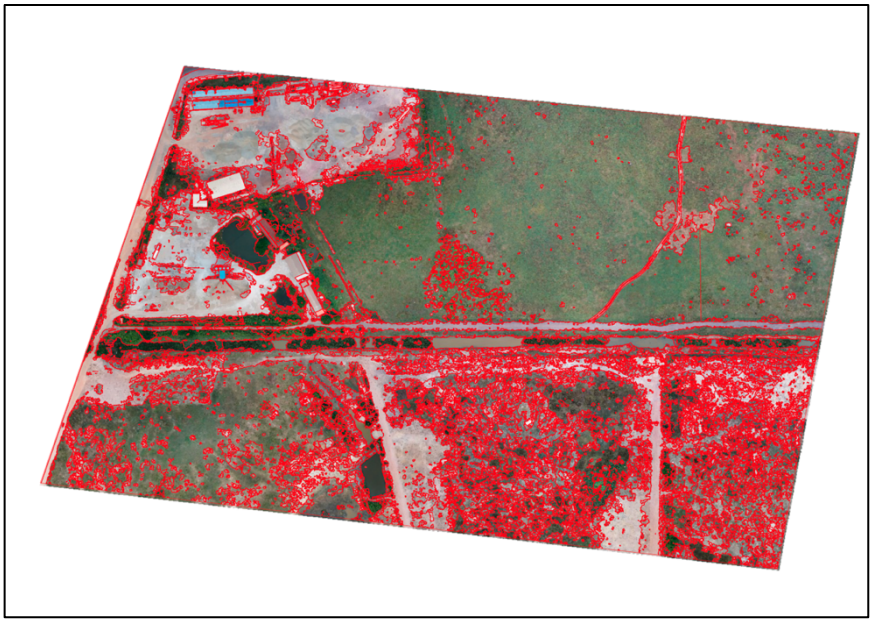


Figure 15.Validation site segmented using Orfeo-Tool Box

The findings indicated that the tiled approach was more effective, as the mean-shift segmentation led to greater overprediction of plastic. This is likely due to the significant variation in the colour and texture of plastics, which the mean-shift algorithm may not adequately capture, as it focuses on spectral homogeneity. Therefore, while the mean-shift algorithm is a powerful segmentation tool, it may not be suitable for high-altitude drone imagery, where assessing plastic accurately depends on texture as well as spectral characteristics.

9. Extended Results – Confusion Matrices

To evaluate model performance and understand classification errors, confusion matrices were generated for both RF and SVM models during training and application. This analysis highlights the models' strengths and weaknesses in distinguishing between different land classes, providing insights into omission and commission errors. Tables 9 displays the confusion matrix generated during the training of the models. Table 10 displays a confusion

matrix generated when applying the model to the validation site. Further analysis of the application confusion matrix is detailed below.

SVM Model Confusion Matrix	Plastic Predicted	Vegetation Predicted	Water Predicted	Earth Predicted	Road Predicted	Building Predicted
Plastic Observed	15	0	0	0	0	1
Vegetation Observed	0	14	0	0	0	0
Water Observed	0	0	18	0	0	0
Earth Observed	0	1	0	13	1	2
Road Observed	1	0	0	0	14	0
Building Observed	0	1	1	2	0	11

RF Model Confusion Matrix	Plastic Predicted	Vegetation Predicted	Water Predicted	Earth Predicted	Road Predicted	Building Predicted
Plastic Observed	15	0	0	0	0	1
Vegetation Observed	0	12	0	1	0	1
Water Observed	0	0	18	0	0	0
Earth Observed	2	0	1	14	0	0
Road Observed	2	0	0	0	13	0
Building Observed	0	1	0	1	2	11

Table 9. Confusion Matrices generated from training data

SVM Model Confusion Matrix	Plastic Predicted	Vegetation Predicted	Water Predicted	Earth Predicted	Road Predicted	Building Predicted
Plastic Observed	13	0	0	13	0	0
Vegetation Observed	20	0	0	545	1	0
Water Observed	0	0	0	6	26	0
Earth Observed	25	0	0	187	30	0
Road Observed	9	0	0	35	38	0
Building Observed	4	0	0	17	10	0

RF Model Confusion Matrix	Plastic Predicted	Vegetation Predicted	Water Predicted	Earth Predicted	Road Predicted	Building Predicted
Plastic Observed	11	1	1	7	0	6
Vegetation Observed	5	372	4	161	7	17
Water Observed	0	0	24	4	0	4
Earth Observed	12	20	1	138	57	14
Road Observed	2	6	2	18	47	11
Building Observed	2	0	2	5	6	16

Table 10. Confusion matrices generated during model application assessment

When analysing commission and omission errors for the plastic class, in the application results, it is important to recognise that the distribution of land classes in the validation site was uneven, with a significantly higher presence of vegetation and earth compared to other classes (Table 6). This imbalance impacts the perception of commission errors. Therefore,

future studies should select test images with a more balanced representation of all land classes or use multiple testing images to obtain more accurate and generalised results.

The SVM model demonstrated a relatively high recall for plastic prediction, correctly identifying plastic 50% of the time, while the other 50% was consistently misclassified as earth. However, the precision for plastic prediction was low, with the model generating many commission errors for the plastic class, particularly in the vegetation (20/71) and earth (25/71) classes.

In comparison, the RF model had a lower recall and frequently confused plastic occurrences with building occurrences (6/26) and earth (7/26) In terms of precision, the RF model was superior to the SVM model, but still had issues, often predicting plastic where there was earth (12/32).

10. Calculations for Scalability

A key objective of the study was to assess the scalability of the project. Table 11 provides detailed estimates of the time taken for each step of the project, which were used to inform the scalability calculations presented in the research paper. This study suggests using the NVIDIA RTX 3080 GPU for processing, comparing its performance to the Apple M2 chip. It is estimated that the RTX 3080 would offer approximately 10 times the processing speed of the Apple M2. This assumption is based on the significantly higher processing power of the RTX 3080, which features 8704 CUDA cores, and its increased memory bandwidth compared to the Apple M2 (Technical City, 2024; GPU Monkey, 2024).

Step	Time/Area	Source	Justification
Drone Imagery Collection	1 day (20km^2)	Report for the UAV Imagery – World Bank project	To collect drone imagery over 20 km^2; typically completed within one day.
Drone Postprocessing	72 hours (20km^2)	Report for the UAV Imagery – World Bank project	Using Agisoft software, processing a 20 km^2 block takes long 24+ hour processing times; some processing took up to one week, hence this study assumes 3 days for typical processing.
Field Survey Ground Data	4.5 hours (~13km^2)	June 2024 Plastic Survey – Green Vientiane and National University of Laos	Conducting 100 survey points across the Bam Hof Agricultural region took 4.5 hours in total.
Prepare Drone Imagery for Model	6 hours (0.25km^2)	Primary data collected by author	This information was noted during the assessment of the models on the 'Validation Site'. Based on the use of a standard laptop (Apple M2, 8GB RAM).
Apply Classification Model	10 hours (0.25km^2)	Primary data collected by author	Same as above.
Stitch Tiles with Classification	15 hours (0.25km^2)	Primary data collected by author	Same as above.

Table 11. Steps and sources of information for the calculations for scalability.

11. Recommendations for Future Research

The results from this initial exploratory study have provided several insights into the methods for detecting and classifying plastic waste piles in Vientiane. This section offers a series of recommendations for future research aimed at improving data sources, training data, model development, and future model outputs to enhance the successful detection and classification of plastic waste.

11.1 Data Sources

- This study used UAV imagery flown at height 400m resulting in a GSD of 5.2cm. Whilst larger plastic waste piles (~400m^2) are clearly identifiable in the image, many smaller sporadic piles exist. These piles without ground truth data are often difficult to distinguish from their surroundings or may be confused with other land classes. Therefore, it is recommended to fly drones at a lower altitude(100-200m) rather than 400m to improve detection accuracy.
- It is recommended to use a multispectral sensor. This study made use of existing RGB drone imagery and highlighted the importance of optical characteristics when characterising land classes. Therefore, expanding spectral band data could prove highly beneficial. Iordashe (2022), Toupuzelis (2019) and Tasserson (2021) highlight the importance of SWIR and NIR bands when detecting plastic.
- The ground data from the pLitter survey and the UAV imagery did not align exactly in date. For the ThatLuang economic zone region, the drone flight was carried out on November 11th, 2021, while the majority of pLitter surveys were conducted in the last two weeks of November 2021. This discrepancy introduces the potential for variation in plastic presence over this period. Therefore, future studies should aim for closer temporal alignment to improve ground truth data accuracy.

11.2 Training Data

- This study demonstrated the challenges of commission errors for the building class. Therefore, masking buildings out of the analysis, similar to the approach used by Ulloa-Torrealba (2023), could mitigate the misclassifications between building and plastic classes.
- Including more land classes to reduce inter-correlation between classes and create more distinct categories could be beneficial (FAO, 2023). For example, the 'Earth' category could be divided to cover agricultural land, shrubland, soil, and clay grounds.
- Additionally, introducing classes for different types of waste, particularly construction waste, is recommended as current models could not differentiate these effectively.

11.3 Model Development

- Further testing of texture characteristics is recommended for smaller-scale studies. Expanding Haralick texture calculations to multiple angles and distances could yield interesting results (Haralick, 1973).
- When assessing the most appropriate parameters for RF and SVM models, such as the number of trees and features for the RF model, future research should test a few

combinations on a small segment of the target image rather than only on training data. This approach would provide a better understanding of model performance in real-world scenarios.

- This study used a more pixel-based classification approach, tiling the drone imagery into 14x14 pixels for model application. Future studies may wish to explore an object-orientated approach. If able to segment the image based on both textural and optical characteristics, rather than just optical, this approach may have significant potential, as it includes further information about plastic pile area, form, shape, texture and context (Xiaoxia et al., 2005).

11.4 Model Outputs

- Once the model has been refined and improved, it is recommended to analyse the typical locations of plastic waste piles based on neighbouring pixels. For instance, determining the prominence of plastic waste in earth, vegetation, and residential (building) areas could provide valuable insights.
- With accurate distribution maps of plastic locations, this research could be effectively integrated with the work of Tran-Thanh et al. (2022) to precisely map plastics at risk of leakage into the Mekong River.

Technical Report – References

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5-32. https://doi.org/10.1023/A:1010933404324

Chen, Z., Wang, L., Wu, W., Jiang, Z., & Li, H. (2016). Monitoring Plastic-Mulched Farmland by Landsat-8 OLI Imagery Using Spectral and Textural Features. Remote Sensing, 8(4), p.353.

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning, 20*(3), 273-297. https://doi.org/10.1007/BF00994018.

Cortesi, I., Masiero, A., De Giglio, M., Tucci, G., & Dubbini, M. (2021). Random Forest-Based River Plastic Detection with a Handheld Multispectral Camera. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, XLIII-B1-2021*, pp.9-14. https://doi.org/10.5194/isprs-archives-XLIII-B1-2021-9-2021

Dalal, N., & Triggs, B. (2005). Histograms of Oriented Gradients for Human Detection. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 1, 886-893. https://doi.org/10.1109/CVPR.2005.177

FAO. (2023). *Using artificial intelligence to assess FAO's knowledge base on the technology accelerator*. Rome: Food and Agriculture Organization of the United Nations. Available at: https://doi.org/10.4060/cc6724en

Feng, Q., Liu, J., & Gong, J. (2015). UAV remote sensing for urban vegetation mapping using random forest and texture analysis. *Remote Sensing*, 7(1), 1074-1094. https://doi.org/10.3390/rs70101074

Frazier, A.E. & Hemingway, B.L. (2021). A Technical Review of Planet Smallsat Data: Practical Considerations for Processing and Using PlanetScope Imagery. *Remote Sensing*, 13(19), 3930. https://doi.org/10.3390/rs13193930

GeeksforGeeks. (2024). How to Avoid Overfitting in SVM. *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/how-to-avoid-overfitting-in-svm/ (Accessed 8 Aug. 2024).

GPU Monkey. (2024). *NVIDIA GeForce RTX 3080 Specifications*. Available at: https://www.gpu-monkey.com/en/#google_vignette (Accessed: 8 August 2024).

Haralick, R.M, Shanmugam, K., Dinstein, I. (1973). Textural Features for Image Classification. IEE Transactions on Systems, Man and Cybernetics, SMC-3, 6, pp.610-621.

Iordache, M.-D., De Keukelaere, L., Moelans, R., Landuyt, L., Moshtaghi, M., Corradi, P., & Knaeps, E. (2022). Targeting Plastics: Machine Learning Applied to Litter Detection in Aerial Multispectral Images. *Remote Sensing*, 14(22), 5820. https://doi.org/10.3390/rs14225820

Karakus, O. (2023). On advances, challenges and potentials of remote sensing image analysis in marine debris and suspected plastics monitoring. Remote Sensing, 4

Kalonde, P. (2022). Geospatial Methods for Mapping Domestic Waste Piles and Macro Plastics. *Master's Thesis*, St. Cloud State University, Department of Geography and Planning. Available at: https://repository.stcloudstate.edu/gp_etds/13

Lary, D.J., Alavi, A.H., Gandomi, A.H. and Walker, A.L., 2016. Machine learning in geosciences and remote sensing. *Geoscience Frontiers*, 7(1), pp.3-10. Available at: https://doi.org/10.1016/j.gsf.2015.07.003 (Accessed 8 Aug. 2024)

Liaw, A. and Wiener, M., 2002. *Classification and regression by randomForest*. R News, 2(3), pp.18-22. Available at: http://www.stat.berkeley.edu/users/breiman/Using_random_forests_V3.1.pdf

Melgani, F. and Bruzzone, L. (2004) Classification of Hyperspectral Remote Sensing Images with Support Vector Machines. IEEE Transactions on Geoscience and Remote Sensing, 42, 1778-1790.

Mifdal, J., Longépé, N. and Rußwurm, M. (2021). Towards Detecting Floating Objects on a Global Scale with Learned Spatial Features Using Sentinel 2, ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, V-3-2021, pp.285–293. doi: https://doi.org/10.5194/isprs-annals-v-3-2021-285-2021.

Orfeo Toolbox. (2024). *Segmentation Application*. Available at: https://www.orfeo-toolbox.org/CookBook-develop/Applications/app_Segmentation.html (Accessed: 10 June 2024).

Planet Labs. (2024). *Sensors Overview*. Available at: https://developers.planet.com/docs/apis/data/sensors/ (Accessed: 9 August 2024).

Sakti, A.D., Sembiring, E., Rohayani, P., Fauzan, K.N., Anggraini, T.S., Santoso, C., Patricia, V.A., Nur, T., Ramadan, A.H., Arjasakusuma, S. and Candra, D.S. (2023). Identification of illegally dumped plastic waste in a highly polluted river in Indonesia using Sentinel-2 satellite imagery. Scientific Reports, 13(1). doi: https://doi.org/10.1038/s41598-023-32087-5.

Sheykhmousa, M., Mahdianpari, M., Ghanbari, H., Mohammadimanesh, F., Ghamisi, P., & Homayouni, S. (2020). Support Vector Machine Versus Random Forest for Remote Sensing Image Classification: A Meta-Analysis and Systematic Review. IEEE Journal of Selected Topics

in Applied Earth Observations and Remote Sensing, 13, 6308-6324. https://doi.org/10.1109/JSTARS.2020.3026724

Tasseron, P., van Emmerik, T., Peller, J., Schreyers, L., & Biermann, L. (2021). Advancing Floating Macroplastic Detection from Space Using Experimental Hyperspectral Imagery. Remote Sensing, 13(12), 2335. https://doi.org/10.3390/rs13122335

Technical City. (2024). *GeForce RTX 3080 vs Apple M2 Max 30-Core GPU*. Available at: https://technical.city/en/video/GeForce-RTX-3080-vs-Apple-M2-Max-30-Core-GPU (Accessed: 8 August 2024).

Topouzelis, K., Papakonstantinou, A., & Garaba, S.P. (2019). Detection of floating plastics from satellite and unmanned aerial systems (Plastic Litter Project 2018). International Journal of Applied Earth Observation and Geoinformation, 79, 175-183. https://doi.org/10.1016/j.jag.2019.03.011

Tran-Thanh, D., Rinasti, A.N., Gunasekara, K., Chaksan, A. and Tsukiji, M. (2022). GIS and Remote Sensing-Based Approach for Monitoring and Assessment of Plastic Leakage and Pollution Reduction in the Lower Mekong River Basin. *Sustainability*, 14(13), p.7879. doi: https://doi.org/10.3390/su14137879.

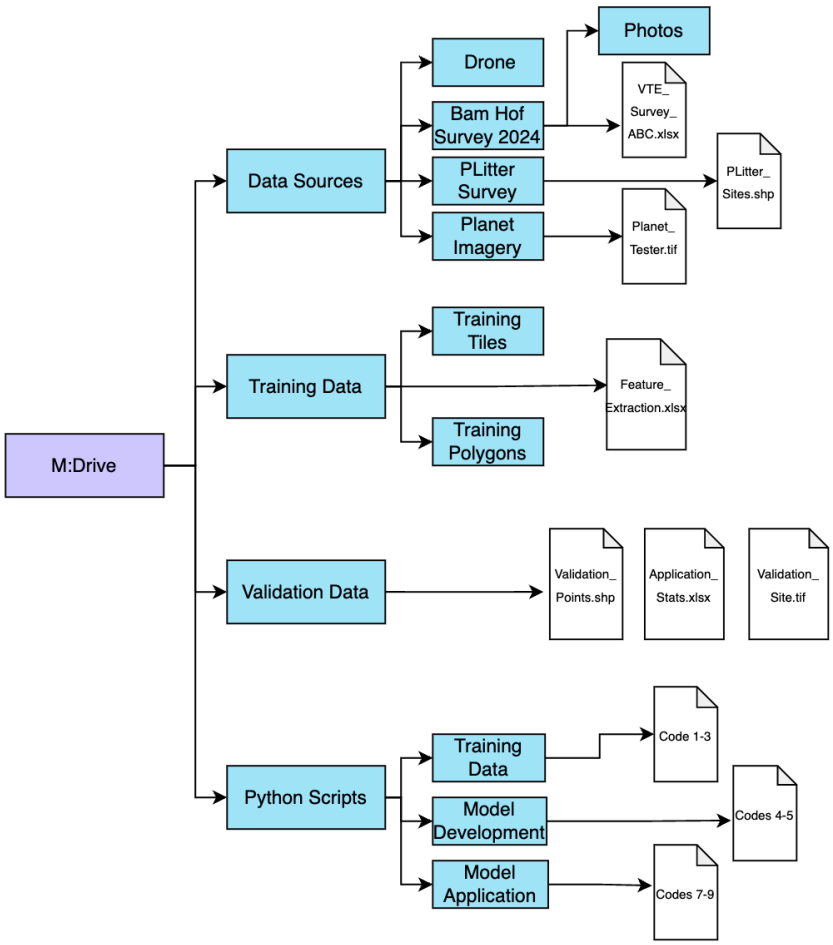
Ulloa-Torrealba, Y.Z., Schmitt, A., Wurm, M., & Taubenböck, H. (2023). Litter on the streets - solid waste detection using VHR images. *European Journal of Remote Sensing*, 56(1), 2176006. https://doi.org/10.1080/22797254.2023.2176006

Valman, S.J., Boyd, D.S., Carbonneau, P.E., Johnson, M.F., and Dugdale, S.J. (2024) *'An AI approach to operationalise global daily PlanetScope satellite imagery for river water masking.' Remote Sensing of Environment,* 301, p.113932. Available at: https://doi.org/10.1016/j.rse.2023.113932 (Accessed: 10 Aug 2024)

Xiaoxia, S., Jixian, Z. and Zhengjun, L., 2005. A comparison of object-oriented and pixel-based classification approaches using Quickbird imagery. *Proceedings of the ISPRS XXXVI 2/W25*, Beijing, China, pp.1-3.

Appendix

M:Drive Folder Layout



Code

This section provides the essential code used for critical steps in the research paper: generation of training data, feature extraction and characterisation, model training, and model application. All code, with exception of training data polygons, was executed in the Python terminal in Visual Studio Code.

Code 1: Extracting training data from QGIS

- Training data was extracted using the Python Console in QGIS
- File name: *Extract_training_data.py*

```
import os
from qgis.core import (
    QgsProject,
    QgsVectorLayer,
    QgsRasterLayer,
    QgsProcessingFeatureSourceDefinition,
    QgsProcessingParameterRasterDestination,
    QgsCoordinateReferenceSystem,
    QgsProcessingContext,
    QgsProcessingFeedback
)
import processing

# Define the paths
vector_layer_path =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/Texture/Training_Data.s
hp'
raster_layer_path =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/Texture/ThatLake.tif'
output_directory =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/Texture/clipped_rasters
'

# Load layers
vector_layer = QgsVectorLayer(vector_layer_path, 'vector_layer', 'ogr')
raster_layer = QgsRasterLayer(raster_layer_path, 'raster_layer')

# Check if the vector layer is valid
if not vector_layer.isValid():
    print("Failed to load vector layer!")
else:
    print("Vector layer loaded successfully!")

# Check if the raster layer is valid
```

```
if not raster_layer.isValid():
    print("Failed to load raster layer!")
else:
    print("Raster layer loaded successfully!")

# Proceed if both layers are valid
if vector_layer.isValid() and raster_layer.isValid():
    # Set output CRS to match raster CRS
    output_crs = raster_layer.crs().authid()

    # Create processing context and feedback objects
    context = QgsProcessingContext()
    feedback = QgsProcessingFeedback()

    # Iterate over each polygon feature
    for feature in vector_layer.getFeatures():
        id = feature.id()  # Get feature ID

        # Define output file path
        output_file = os.path.join(output_directory, f'clipped_raster_{id}.tif')

        # Create a temporary layer for the single feature
        temp_layer =
QgsVectorLayer("Polygon?crs={}".format(vector_layer.crs().authid()), "temp_layer",
"memory")
        temp_layer_data = temp_layer.dataProvider()
        temp_layer_data.addAttributes(vector_layer.fields())
        temp_layer.updateFields()
        temp_layer_data.addFeature(feature)

        # Define clipping extent and options
        params = {
            'INPUT': raster_layer_path,
            'MASK': temp_layer,
            'SOURCE_CRS': raster_layer.crs(),
            'TARGET_CRS': output_crs,
            'NO_DATA': 0,  # Set the NoData value (adjust as needed)
            'ALPHA_BAND': True,
            'CROP_TO_CUTLINE': True,
            'KEEP_RESOLUTION': True,
            'OPTIONS': '',
            'DATA_TYPE': 0,
            'OUTPUT': output_file
        }

       try:
            # Run the processing algorithm
            result = processing.run("gdal:cliprasterbymasklayer", params,
context=context, feedback=feedback)
            if result['OUTPUT']:
                print(f'Processed polygon with ID: {id}')
```

```
            else:
                print(f'Failed to process polygon with  ID: {id}')

        except Exception as e:
            print(f'Failed to process polygon with ID: {id}, error: {e}')

    print("Processing completed!")
else:
    print("Layers are not valid, cannot proceed.")
```

Code 2: Creating training data tiles

- This code takes the training data polygons from *Code 1* and creates 28x28 pixel tiles from each polygon
- File name: *TiledUp_trainingData.py*

```
import os
import rasterio
import numpy as np

# Define the directories and range for each category
categories = {
    "water": range(1, 61),
    "earth": range(61, 121),
    "building": range(121, 181),
    "roads": range(181, 241),
    "vegetation": range(241, 301),
    "plastic": range(301, 361)
}

input_dir =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/TrainingData_0207'
output_base_dir =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/TiledRasterOutput2'

# Create output directories
for category in categories.keys():
    os.makedirs(os.path.join(output_base_dir, f'tiledupraster_{category}'),
exist_ok=True)

def process_raster(file_path, output_dir):
    with rasterio.open(file_path) as src:
        for i in range(0, src.height, 28):
            for j in range(0, src.width, 28):
                # Calculate window
                window = rasterio.windows.Window(j, i, 28, 28)
                # Read window data
```

```
                tile = src.read(window=window)
                if tile.shape[1] == 28 and tile.shape[2] == 28:
                    tile_file_name =
f"{os.path.basename(file_path).split('.')[0]}_{i}_{j}.tif"
                    tile_file_path = os.path.join(output_dir, tile_file_name)

                    # Save the tile
                    with rasterio.open(
                        tile_file_path, 'w',
                        driver='GTiff',
                        height=28, width=28,
                        count=src.count,
                        dtype=tile.dtype,
                        crs=src.crs,
                        transform=src.window_transform(window)
                    ) as dst:
                        dst.write(tile)

for category, file_range in categories.items():
    output_dir = os.path.join(output_base_dir, f'tiledupraster_{category}')
    for file_index in file_range:
        file_name = f'clipped_raster_{file_index}.tif'
        file_path = os.path.join(input_dir, file_name)
        if os.path.exists(file_path):
            process_raster(file_path, output_dir)
        else:
            print(f"File {file_name} not found in {input_dir}")
```

Code 3: Calculating feature characteristics from the training tiles

- Calculates optical and textural properties of the training tiles into an excel file. The features calculated here are the ones for the final models.
- File Name: *features1101.py*

```
import os
import pandas as pd
from osgeo import gdal
import numpy as np
from skimage.feature import graycomatrix, graycoprops
from scipy.stats import skew, entropy
import cv2

def load_image(file_path):
    """
    Loads an image using GDAL and reads its bands as arrays.

    Argument:
        file_path (str): Path to the image file.

    Returns:
        list: A list of arrays, each representing a band of the image.
```

```
    """
    dataset = gdal.Open(file_path)
    bands = [dataset.GetRasterBand(i+1).ReadAsArray() for i in
range(dataset.RasterCount)]
    return bands

def compute_rgb_features(bands):
    """
    Computes the mean and standard deviation for the RGB bands of an image.

    Argument:
        bands (list): A list of arrays representing the bands of the image.

    Returns:
        dict: A dictionary containing the mean and standard deviation of the RGB
bands.
    """
    red_band = bands[0]
    green_band = bands[1]
    blue_band = bands[2]

    # Compute mean
    mean_red = np.mean(red_band)
    mean_green = np.mean(green_band)
    mean_blue = np.mean(blue_band)

    # Compute standard deviation
    std_red = np.std(red_band)
    std_green = np.std(green_band)
    std_blue = np.std(blue_band)

    return {
        'mean_red': mean_red, 'mean_green': mean_green, 'mean_blue': mean_blue,
        'std_red': std_red, 'std_green': std_green, 'std_blue': std_blue,
    }

def compute_hsv_features(bands):
    """
    Computes the mean and standard deviation for the HSV representation of an
image.

    Argument:
        bands (list): A list of arrays representing the bands of the image.

    Returns:
        dict: A dictionary containing the mean and standard deviation of the HSV
channels.
    """
    rgb_image = np.stack(bands, axis=-1).astype(np.uint8)
    hsv_image = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2HSV)
```

```
    mean_hue = np.mean(hsv_image[:, :, 0])
    mean_saturation = np.mean(hsv_image[:, :, 1])
    mean_value = np.mean(hsv_image[:, :, 2])

    std_hue = np.std(hsv_image[:, :, 0])
    std_saturation = np.std(hsv_image[:, :, 1])
    std_value = np.std(hsv_image[:, :, 2])

    return {
        'mean_hue': mean_hue, 'mean_saturation': mean_saturation, 'mean_value':
mean_value,
        'std_hue': std_hue, 'std_saturation': std_saturation, 'std_value':
std_value,
    }

def compute_extended_texture_features(glcm):
    """
    Computes extended texture features from the Gray Level Co-occurrence Matrix
(GLCM).

    Argument:
        glcm (ndarray): The Gray Level Co-occurrence Matrix.

    Returns:
        dict: A dictionary containing various texture features.
    """
    properties = {
        'entropy': lambda P: entropy(P.ravel()),
        'inverse_difference_moment': lambda P: np.sum(P / (1. +
(np.indices(P.shape)[0] - np.indices(P.shape)[1]) ** 2)),
        'cluster_shade': lambda P: np.sum((np.indices(P.shape)[0] +
np.indices(P.shape)[1] - 2 * np.mean(P)) ** 3 * P),
        'cluster_prominence': lambda P: np.sum((np.indices(P.shape)[0] +
np.indices(P.shape)[1] - 2 * np.mean(P)) ** 4 * P),
    }

    feature_values = {name: func(glcm) for name, func in properties.items()}
    return feature_values

def compute_haralick_texture(band, window_size=5):
    """
    Computes Haralick texture features for a given band of an image.

    Argument:
        band (ndarray): The image band array.
        window_size (int): The size of the sliding window.

    Returns:
        dict: A dictionary containing the average Haralick texture features.
    """
    height, width = band.shape
```

```
    texture_features = []

    for i in range(0, height - window_size + 1, window_size):
        for j in range(0, width - window_size + 1, window_size):
            window = band[i:i + window_size, j:j + window_size]
            distances = [1]
            angles = [0]
            glcm = graycomatrix(window, distances=distances, angles=angles,
levels=256, symmetric=True, normed=True)
            properties = ['contrast', 'dissimilarity', 'homogeneity']
            textures = [graycoprops(glcm, prop).ravel()[0] for prop in properties]
            extended_textures = compute_extended_texture_features(glcm)
            texture_features.append(textures + list(extended_textures.values()))

    # Average texture features over all windows
    if texture_features:
        avg_textures = np.mean(texture_features, axis=0)
        all_properties = properties + list(extended_textures.keys())
        texture_dict = {prop: avg_textures[idx] for idx, prop in
enumerate(all_properties)}
        return texture_dict
    else:
        return {prop: None for prop in properties}

def process_image(file_path):
    """
    Processes an image to extract RGB, HSV, and texture features.

    Argument:
        file_path (str): Path to the image file.

    Returns:
        dict: A dictionary containing all the extracted features.
    """
    bands = load_image(file_path)

    if len(bands) < 3:
        raise ValueError("Image does not have enough bands (RGB) to process.")

    rgb_features = compute_rgb_features(bands[:3])
    hsv_features = compute_hsv_features(bands[:3])

    texture_features = {}
    for i, band in enumerate(bands[:3]):
        band_textures = compute_haralick_texture(band)
        for k, v in band_textures.items():
            avg_key = f'avg_{k}'
            if avg_key in texture_features:
                texture_features[avg_key].append(v)
            else:
                texture_features[avg_key] = [v]
```

```
    # Average the texture features across RGB bands
    avg_texture_features = {k: np.mean(v) for k, v in texture_features.items()}

    return {**{'file_name': os.path.basename(file_path)}, **rgb_features,
**hsv_features, **avg_texture_features}

def process_all_images(folder_path, output_excel, batch_size=120):
    """
    Processes all images in a folder to extract features and saves the results to
an Excel file.

    Argument:
        folder_path (str): Path to the folder containing image files.
        output_excel (str): Path to the output Excel file.
        batch_size (int): Number of files to process in each batch.
    """
    files = [os.path.join(folder_path, f) for f in os.listdir(folder_path) if
f.endswith('.tif')]
    all_data = []
    processed_files = set()

    if os.path.exists(output_excel):
        df_existing = pd.read_excel(output_excel)
        processed_files = set(df_existing['file_name'].tolist())
        all_data = df_existing.to_dict(orient='records')

    for i in range(0, len(files), batch_size):
        batch_files = files[i:i + batch_size]
        batch_data = []

        for file_path in batch_files:
            file_name = os.path.basename(file_path)
            if file_name in processed_files:
                print(f"Skipping already processed file: {file_name}")
                continue

            try:
                result = process_image(file_path)
                batch_data.append(result)
                processed_files.add(file_name)
            except Exception as e:
                print(f"Error processing {file_path}: {e}")

        all_data.extend(batch_data)
        df = pd.DataFrame(all_data)
        df.to_excel(output_excel, index=False)
        print(f"Saved batch {i // batch_size + 1} results to {output_excel}")

# Main execution
```

```
folder_path =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/TrainingData_0207'
output_excel = os.path.join(folder_path, 'all_characteristics_1007_v1.xlsx')
process_all_images(folder_path, output_excel, batch_size=20)
```

Code 4: Training Random Forest model

- Trains model using the training data feature data generated in *Code 3*
- Feature importance and proximity matrix also generated
- File name: *FinalRFmodel_LC.py*

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
import seaborn as sns
import joblib

"""
This script performs the following tasks:
1. Loads a dataset from an Excel file and prepares it by selecting specific
features and the target variable.
2. Splits the data into training and testing sets with stratification.
3. Trains a Random Forest Classifier and evaluates its accuracy.
4. Identifies the most important features for predicting the different land
classes.
5. Plots the feature importance as a bar chart with the top 10 features, using
different colours for optical and texture features.
6. Calculates and visualizes the proximity matrix to understand the similarity
between land classes.
7. Saves the trained model to a file and demonstrates loading the model for
predictions.
"""

# Load the dataset
file_path =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/NEWtrainingData/NewData
_Characteristics_v2.xlsx'
data = pd.read_excel(file_path)

# Drop 'file_name' and 'subfolder' columns
data = data.drop(columns=['file_name', 'subfolder'])
```

```
# Set the target column
target_column = 'Land Class'

# Define the features to be used
features = [
    'mean_red', 'mean_green', 'mean_blue',
    'std_red', 'std_green', 'std_blue',
    'mean_hue', 'mean_saturation', 'mean_value',
    'std_hue', 'std_saturation', 'std_value',
    'avg_contrast', 'avg_dissimilarity', 'avg_homogeneity',
    'avg_entropy', 'avg_inverse_difference_moment',
    'avg_cluster_shade', 'avg_cluster_prominence'
]

# Data preparation
X = data[features]  # Select only the specified features
y = data[target_column]  # Target variable

# Split the data into training and testing sets with stratification
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)

# Train the final model with chosen hyperparameters
final_rf_model = RandomForestClassifier(n_estimators=30, max_features=12,
random_state=42, oob_score=True)
final_rf_model.fit(X_train, y_train)

# Evaluate the final model
y_pred = final_rf_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f'Overall Accuracy: {accuracy:.2f}')

# Classification report
report = classification_report(y_test, y_pred)
print(f"\nClassification Report:\n{report}")

# Feature Importance for predicting different land classes
importances = final_rf_model.feature_importances_
feature_names = X.columns
feature_importance_df = pd.DataFrame({'Feature': feature_names, 'Importance':
importances})

# Sort the features by importance and select top 10
feature_importance_df = feature_importance_df.sort_values(by='Importance',
ascending=False).head(10)

# Define hatch patterns for color and texture features
optical_features = [
    'mean_red', 'mean_green', 'mean_blue',
    'std_red', 'std_green', 'std_blue',
    'mean_hue', 'mean_saturation', 'mean_value',
```

```
    'std_hue', 'std_saturation', 'std_value'
]
texture_features = [
    'avg_contrast', 'avg_dissimilarity', 'avg_homogeneity',
    'avg_entropy', 'avg_inverse_difference_moment',
    'avg_cluster_shade', 'avg_cluster_prominence'
]

# Plot the feature importance using matplotlib
plt.figure(figsize=(10, 8))
bars = plt.barh(
    feature_importance_df['Feature'],
    feature_importance_df['Importance'],
    color=['darkgrey' if feature in optical_features else 'lightgrey' for feature
in feature_importance_df['Feature']],
    edgecolor='black'
)

plt.xlabel('Importance')
plt.title('Top 10 Feature Importances for Predicting Land Classes')
plt.gca().invert_yaxis()  # Invert y-axis to have the highest importance on top

# Annotate the bars with the importance values
for bar in bars:
    plt.text(bar.get_width(), bar.get_y() + bar.get_height()/2,
f'{bar.get_width():.4f}', va='center', ha='left')

plt.show()

# Calculate proximity matrix between land classes
unique_classes = y_train.unique()
class_proximity_matrix = np.zeros((len(unique_classes), len(unique_classes)))

# Create a mapping from class to index
class_to_index = {cls: idx for idx, cls in enumerate(unique_classes)}

# Loop through each tree in the random forest
for tree in final_rf_model.estimators_:
    leaf_indices = tree.apply(X_train)
    for i in range(X_train.shape[0]):
        for j in range(i, X_train.shape[0]):
            if leaf_indices[i] == leaf_indices[j]:
                class_i = y_train.iloc[i]
                class_j = y_train.iloc[j]
                index_i = class_to_index[class_i]
                index_j = class_to_index[class_j]
                class_proximity_matrix[index_i, index_j] += 1
                if index_i != index_j:
                    class_proximity_matrix[index_j, index_i] += 1

# Normalize the proximity matrix by the number of trees and samples in each class
```

```
class_proximity_matrix /= final_rf_model.n_estimators

# Apply logarithmic scaling to the proximity matrix
log_class_proximity_matrix = np.log1p(class_proximity_matrix)

# Visualize proximity matrix using a heatmap with a logarithmic scale
plt.figure(figsize=(10, 8))
sns.heatmap(log_class_proximity_matrix, cmap='viridis', xticklabels=unique_classes,
yticklabels=unique_classes, linewidths=1, linecolor='black')
plt.title('Logarithmic Class Proximity Matrix Heatmap')
plt.xlabel('Classes')
plt.ylabel('Classes')
plt.show()

# Save the model to a file
model_filename =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/Texture/RF_model_LC_vI.
joblib'
joblib.dump(final_rf_model, model_filename)
print(f'Model saved to {model_filename}')

# Load the model from the file
loaded_model = joblib.load(model_filename)

# Use the loaded model to make predictions
y_pred_loaded = loaded_model.predict(X_test)
accuracy_loaded = accuracy_score(y_test, y_pred_loaded)
print(f'Loaded model accuracy: {accuracy_loaded:.2f}')
```

Code 5: Training Support Vector Model

- Trains model using a subset of the training data feature data generated in *Code 3.* This is as generating the SVM model was more computationally intensive, hence chose to include the top ten features for the model.
- File name: *FinalSVMmodel_LC.py*

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report,
cohen_kappa_score, confusion_matrix
import joblib

"""
This script performs the following tasks:
```

```
1. Loads a dataset from an Excel file and prepares it by selecting specific
features and the target variable.
2. Splits the data into training and testing sets with stratification.
3. Trains an SVM model with different hyperparameters and evaluates its accuracy.
4. Saves the best model based on predefined hyperparameters.
5. Plots the confusion matrix for the saved model.
6. Plots a heatmap of the accuracy results from the hyperparameter grid search.
"""

# Load the dataset
file_path = '/home/s2025580/scratch/Dis0108/NewData_Characteristics_v3.xlsx'
data = pd.read_excel(file_path)
data = data.drop(columns=['file_name', 'subfolder'])

# Set the target column
target_column = 'Land Class'

# Define the features to be used
features = [
    'std_red', 'std_green', 'std_blue',
    'mean_hue', 'mean_saturation',
    'std_hue', 'std_saturation', 'std_value',
    'avg_contrast', 'avg_dissimilarity',
]

# Data preparation
X = data[features]
y = data[target_column]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)

# Train and evaluate the SVM model
param_grid = {
    'C': [10, 1, 0.1],
    'gamma': ['scale', 'auto'],
    'kernel': ['linear', 'rbf']
}
results = []
for C in param_grid['C']:
    for gamma in param_grid['gamma']:
        for kernel in param_grid['kernel']:
            model = SVC(C=C, gamma=gamma, kernel=kernel, random_state=42)
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
            accuracy = accuracy_score(y_test, y_pred)
            results.append((C, gamma, kernel, accuracy))
            if C == 1 and gamma == 'scale' and kernel == 'linear':
                # Save the model with C=1, gamma='scale', and kernel='linear'
                model_filename =
'/home/s2025580/scratch/Dis0108/SVM_C1_scale_linear.joblib'
                joblib.dump(model, model_filename)
```

```
                print(f'Model saved to {model_filename}')

                # Print classification report
                print("\nClassification Report for Model (C=1, gamma='scale',
kernel='linear'):\n")
                print(classification_report(y_test, y_pred))

                # Print Cohen's Kappa
                kappa = cohen_kappa_score(y_test, y_pred)
                print(f"\nCohen's Kappa: {kappa:.2f}")

                # Print confusion matrix
                cm = confusion_matrix(y_test, y_pred, labels=model.classes_)
                cm_df = pd.DataFrame(cm, index=model.classes_,
columns=model.classes_)
                plt.figure(figsize=(10, 8))
                sns.heatmap(cm_df, annot=True, cmap="viridis", fmt="d")
                plt.title('Confusion Matrix for Model (C=1, gamma="scale",
kernel="linear")')
                plt.ylabel('Actual')
                plt.xlabel('Predicted')
                plt.show()

# Create a DataFrame to store the results
results_df = pd.DataFrame(results, columns=['C', 'Gamma', 'Kernel', 'Accuracy'])

# Pivot the DataFrame for the heatmap
pivot_table = results_df.pivot_table(index='C', columns=['Gamma', 'Kernel'],
values='Accuracy')

# Plot the heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(pivot_table, annot=True, cmap="viridis", fmt=".2f")
plt.title('Accuracy Heatmap for SVM Hyperparameter Grid Search')
plt.show()

# Load the model from the file (optional demonstration)
loaded_model = joblib.load(model_filename)

# Use the loaded model to make predictions
y_pred_loaded = loaded_model.predict(X_test)
accuracy_loaded = accuracy_score(y_test, y_pred_loaded)
print(f'Loaded model accuracy: {accuracy_loaded:.2f}')
```

Code 6: Creating tiles from drone imagery

- To tile sections of the drone imagery, the process was often carried out in two steps. First, an initial breakdown of the drone imagery into a specified number of larger tiles was performed. Then, these larger tiles were further subdivided into smaller 14x14 tiles. This two-step approach was employed to reduce the computational intensity of creating 14x14 tiles all at once.
- Files names: *Drone_initialtiles.py* and *Drone_tilestiles.py*

```
import rasterio
import numpy as np
import os

def create_tiles(image, num_tiles_x, num_tiles_y, output_dir, batch_size=1000):
    """
    Splits a large image into smaller tiles and saves them in batches.

    Arguments:
        image (str): Path to the input image file.
        num_tiles_x (int): Number of tiles along the width.
        num_tiles_y (int): Number of tiles along the height.
        output_dir (str): Directory to save the tiles.
        batch_size (int): Number of tiles per batch.
    """
    with rasterio.open(image) as src:
        bands = src.read()
        nrows, ncols = bands.shape[1], bands.shape[2]

        tile_width = ncols // num_tiles_x
        tile_height = nrows // num_tiles_y

        tile_num = 0
        batch_num = 0
        batch_dir = os.path.join(output_dir, f"batch_{batch_num}")
        os.makedirs(batch_dir, exist_ok=True)

        for i in range(num_tiles_y):
            for j in range(num_tiles_x):
                x_start, x_end = j * tile_width, (j + 1) * tile_width
                y_start, y_end = i * tile_height, (i + 1) * tile_height

                tile = bands[:, y_start:y_end, x_start:x_end]

                output_path = os.path.join(batch_dir, f"tile_{tile_num}.tif")
                with rasterio.open(
                    output_path,
                    'w',
                    driver='GTiff',
                    height=tile_height,
                    width=tile_width,
```

```
                    count=bands.shape[0],
                    dtype=bands.dtype,
                    crs=src.crs,
                    transform=rasterio.windows.transform(
                        rasterio.windows.Window(x_start, y_start, tile_width,
tile_height),
                        src.transform,
                    ),
                ) as dst:
                    dst.write(tile)

                tile_num += 1
                if tile_num % batch_size == 0:
                    batch_num += 1
                    batch_dir = os.path.join(output_dir, f"batch_{batch_num}")
                    os.makedirs(batch_dir, exist_ok=True)

def main():
    image_path =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/SubsetDroneforModelApp/
Riverdrone/Riverdrone.tif'
    num_tiles_x = 200  # Number of tiles along the width
    num_tiles_y = 500  # Number of tiles along the height
    output_dir =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/SubsetDroneforModelApp/
Riverdrone/RiverDroneTiles'  # Directory to save the tiles
    batch_size = 1000  # Number of tiles per batch

    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    create_tiles(image_path, num_tiles_x, num_tiles_y, output_dir, batch_size)

if __name__ == "__main__":
    main()
```

```
import os
import rasterio
import numpy as np

def create_14x14_tiles(tile_dir, output_dir, batch_size=1000):
    """
    Splits existing tiles into smaller 14x14 tiles and saves them in batches.

    Arguments:
        tile_dir (str): Directory containing the input tiles.
        output_dir (str): Directory to save the smaller tiles.
        batch_size (int): Number of tiles per batch.
    """
    batch_num = 0
```

```
    batch_dir = os.path.join(output_dir, f"batch_{batch_num}_14")
    os.makedirs(batch_dir, exist_ok=True)

    tile_num = 0
    for subdir, _, files in os.walk(tile_dir):
        for tile_file in files:
            if tile_file.endswith('.tif'):
                tile_path = os.path.join(subdir, tile_file)

                with rasterio.open(tile_path) as src:
                    bands = src.read()
                    nrows, ncols = bands.shape[1], bands.shape[2]

                    num_tiles_x = ncols // 14
                    num_tiles_y = nrows // 14

                    for i in range(num_tiles_y):
                        for j in range(num_tiles_x):
                            x_start, x_end = j * 14, (j + 1) * 14
                            y_start, y_end = i * 14, (i + 1) * 14

                            sub_tile = bands[:, y_start:y_end, x_start:x_end]

                            output_path = os.path.join(batch_dir,
f"tile_{tile_num}.tif")
                            with rasterio.open(
                                output_path,
                                'w',
                                driver='GTiff',
                                height=14,
                                width=14,
                                count=bands.shape[0],
                                dtype=bands.dtype,
                                crs=src.crs,
                                transform=rasterio.windows.transform(
                                    rasterio.windows.Window(x_start, y_start, 14,
14),
                                    src.transform,
                                ),
                            ) as dst:
                                dst.write(sub_tile)

                            tile_num += 1
                            if tile_num % batch_size == 0:
                                batch_num += 1
                                batch_dir = os.path.join(output_dir,
f"batch_{batch_num}_14")
                                os.makedirs(batch_dir, exist_ok=True)

def main():
```

```
    input_tile_dir =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/SubsetDroneforModelApp/
Riverdrone/RiverDroneTiles'
    output_dir =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/SubsetDroneforModelApp/
Riverdrone/RiverDroneTiles14'
    batch_size = 1000  # Number of tiles per batch

    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    create_14x14_tiles(input_tile_dir, output_dir, batch_size)

if __name__ == "__main__":
    main()
```

Code 7: Applying the RF model to tiles

- Predictions made for each tile and output into an excel file
- File name: *Apply_RF.tiles*

```
import os
import pandas as pd
import numpy as np
import rasterio
from skimage.feature import graycomatrix, graycoprops
import joblib

"""
This script performs the following tasks:
1. Loads a pre-trained Random Forest (RF) model.
2. Defines functions to compute various features from images, including mean and
standard deviation of RGB channels, mean and standard deviation of HSV channels,
and Haralick texture features.
3. Processes batches of image tiles to extract these features and uses the RF model
to make predictions.
4. Saves the predictions and any errors encountered during processing to an Excel
file and a text log, respectively.
"""

# Load the pre-trained RF model
model_path =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/Texture/RF_model_LC_vH.
joblib'
model_RF = joblib.load(model_path)

# Define the expected feature names
expected_features = [
    'mean_red', 'mean_green', 'mean_blue',
    'std_red', 'std_green', 'std_blue',
    'mean_hue', 'mean_saturation', 'mean_value',
```

```
    'std_hue', 'std_saturation', 'std_value',
    'avg_contrast', 'avg_dissimilarity', 'avg_homogeneity',
    'avg_entropy', 'avg_inverse_difference_moment',
    'avg_cluster_shade', 'avg_cluster_prominence'
]

def compute_mean_rgb(image):
    mean_red = np.mean(image[:, :, 0])
    mean_green = np.mean(image[:, :, 1])
    mean_blue = np.mean(image[:, :, 2])
    return mean_red, mean_green, mean_blue

def compute_std_rgb(image):
    std_red = np.std(image[:, :, 0])
    std_green = np.std(image[:, :, 1])
    std_blue = np.std(image[:, :, 2])
    return std_red, std_green, std_blue

def compute_mean_hsv(image):
    hsv_image = np.array(image).astype(np.float32)
    hsv_image /= 255.0
    mean_hue = np.mean(hsv_image[:, :, 0])
    mean_saturation = np.mean(hsv_image[:, :, 1])
    mean_value = np.mean(hsv_image[:, :, 2])
    return mean_hue, mean_saturation, mean_value

def compute_std_hsv(image):
    hsv_image = np.array(image).astype(np.float32)
    hsv_image /= 255.0
    std_hue = np.std(hsv_image[:, :, 0])
    std_saturation = np.std(hsv_image[:, :, 1])
    std_value = np.std(hsv_image[:, :, 2])
    return std_hue, std_saturation, std_value

def compute_haralick_texture(image):
    properties = ['contrast', 'dissimilarity', 'homogeneity', 'entropy',
'inverse_difference_moment', 'cluster_shade', 'cluster_prominence']
    texture_features = []
    for i in range(1):  # Process the first band (B1) only
        glcm = graycomatrix(image[:, :, i], distances=[1], angles=[0], levels=256,
symmetric=True, normed=True)
        textures = [graycoprops(glcm, prop).ravel()[0] for prop in properties[:3]]
        entropy_value = -np.sum(glcm * np.log2(glcm + (glcm == 0)))
        idm = np.sum(glcm / (1 + np.square(np.arange(glcm.shape[0]) -
np.arange(glcm.shape[1]))[:, None]))
        cluster_shade = np.sum((glcm - np.mean(glcm))**3)
        cluster_prominence = np.sum((glcm - np.mean(glcm))**4)
        texture_features.extend(textures + [entropy_value, idm, cluster_shade,
cluster_prominence])
    return texture_features
```

```
def process_batch(tile_files, tiles_dir, subfolder_name):
    feature_vectors = []
    error_files = []

    for tile_file in tile_files:
        tile_path = os.path.join(tiles_dir, tile_file)

        try:
            # Open the tile, read the data, and then close the dataset
            with rasterio.open(tile_path) as src:
                image = src.read().transpose(1, 2, 0)  # Reorder to (row, col,
band)

            # Compute mean RGB
            mean_red, mean_green, mean_blue = compute_mean_rgb(image)

            # Compute standard deviation RGB
            std_red, std_green, std_blue = compute_std_rgb(image)

            # Compute mean HSV
            mean_hue, mean_saturation, mean_value = compute_mean_hsv(image)

            # Compute standard deviation HSV
            std_hue, std_saturation, std_value = compute_std_hsv(image)

            # Compute Haralick texture features
            texture_features = compute_haralick_texture(image)

            # Create a dictionary for the feature vector with standardized names
            feature_vector = {
                'mean_red': mean_red,
                'mean_green': mean_green,
                'mean_blue': mean_blue,
                'std_red': std_red,
                'std_green': std_green,
                'std_blue': std_blue,
                'mean_hue': mean_hue,
                'mean_saturation': mean_saturation,
                'mean_value': mean_value,
                'std_hue': std_hue,
                'std_saturation': std_saturation,
                'std_value': std_value,
                'avg_contrast': texture_features[0],
                'avg_dissimilarity': texture_features[1],
                'avg_homogeneity': texture_features[2],
                'avg_entropy': texture_features[3],
                'avg_inverse_difference_moment': texture_features[4],
                'avg_cluster_shade': texture_features[5],
                'avg_cluster_prominence': texture_features[6]
            }
```

```
            # Ensure the feature vector matches the expected features
            ordered_feature_vector = [feature_vector[feature] for feature in
expected_features]
            feature_vectors.append(ordered_feature_vector)

        except rasterio.errors.RasterioIOError as e:
            print(f"Error reading {tile_path}: {e}")
            error_files.append(tile_file)
            continue

    return feature_vectors, error_files

def process_all_tiles(base_dir, batch_size=5000):
    """
    Process all tiles in the given directory, batch-wise, to extract features and
predict land classes.

    Args:
        base_dir (str): Base directory containing the tile subdirectories.
        batch_size (int): Number of tiles to process per batch.
    """
    all_feature_vectors = []
    all_predictions = []
    all_tile_files = []
    all_error_files = []

    for subdir, _, files in os.walk(base_dir):
        tile_files = [f for f in files if f.endswith('.tif')]
        subfolder_name = os.path.basename(subdir)
        num_batches = len(tile_files) // batch_size + (1 if len(tile_files) %
batch_size != 0 else 0)

        for batch_idx in range(num_batches):
            start_idx = batch_idx * batch_size
            end_idx = start_idx + batch_size
            batch_files = tile_files[start_idx:end_idx]

            try:
                batch_feature_vectors, error_files = process_batch(batch_files,
subdir, subfolder_name)
                all_feature_vectors.extend(batch_feature_vectors)
                all_error_files.extend(error_files)

                # Convert list of feature vectors to DataFrame
                batch_features_df = pd.DataFrame(batch_feature_vectors,
columns=expected_features)

                # Apply the pre-trained model for class predictions
                X_test = batch_features_df.values  # Ensure the correct order of
features
                batch_predictions = model_RF.predict(X_test)  # Class predictions
```

```
                all_predictions.extend(batch_predictions)
                all_tile_files.extend([(subfolder_name, tile_file) for tile_file in
batch_files])
                print(f'Processed batch {batch_idx + 1}/{num
```

Code 8: Applying the SVM models to tiles

- Predictions made for each tile and output into an excel file
- File name: *Apply_SVM.tiles*

```
import os
import pandas as pd
import numpy as np
import rasterio
from skimage.feature import graycomatrix, graycoprops
import joblib

"""
This script performs the following tasks:
1. Loads a pre-trained SVM model.
2. Defines functions to compute various features from images, including standard
deviation of RGB channels, mean and standard deviation of HSV channels, and
Haralick texture features.
3. Processes batches of image tiles to extract these features and uses the SVM
model to make predictions.
4. Saves the predictions and any errors encountered during processing to an Excel
file and a text log, respectively.
"""

# Load the pre-trained SVM model
model_path =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/Texture/Shell_and_Py/Mo
dels_Complete/Top4/SVM_C1_scale_linear.joblib'
model_SVM = joblib.load(model_path)

# Define the expected feature names
expected_features = [
    'std_red', 'std_green', 'std_blue',
    'mean_hue', 'mean_saturation',
    'std_hue', 'std_saturation', 'std_value',
    'avg_contrast', 'avg_dissimilarity'
]

def compute_std_rgb(image):
    std_red = np.std(image[:, :, 0])
    std_green = np.std(image[:, :, 1])
    std_blue = np.std(image[:, :, 2])
    return std_red, std_green, std_blue
```

```
def compute_mean_hsv(image):
    hsv_image = np.array(image).astype(np.float32)
    hsv_image /= 255.0
    mean_hue = np.mean(hsv_image[:, :, 0])
    mean_saturation = np.mean(hsv_image[:, :, 1])
    return mean_hue, mean_saturation

def compute_std_hsv(image):
    hsv_image = np.array(image).astype(np.float32)
    hsv_image /= 255.0
    std_hue = np.std(hsv_image[:, :, 0])
    std_saturation = np.std(hsv_image[:, :, 1])
    std_value = np.std(hsv_image[:, :, 2])
    return std_hue, std_saturation, std_value

def compute_haralick_texture(image):
    properties = ['contrast', 'dissimilarity']
    texture_features = []
    for i in range(1):  # Process the first band (B1) only
        glcm = graycomatrix(image[:, :, i], distances=[1], angles=[0], levels=256,
symmetric=True, normed=True)
        textures = [graycoprops(glcm, prop).ravel()[0] for prop in properties]
        texture_features.extend(textures)
    return texture_features

def process_batch(tile_files, tiles_dir, subfolder_name):
    feature_vectors = []
    error_files = []

    for tile_file in tile_files:
        tile_path = os.path.join(tiles_dir, tile_file)

        try:
            # Open the tile, read the data, and then close the dataset
            with rasterio.open(tile_path) as src:
                image = src.read().transpose(1, 2, 0)  # Reorder to (row, col,
band)

            # Compute standard deviation RGB
            std_red, std_green, std_blue = compute_std_rgb(image)

            # Compute mean HSV
            mean_hue, mean_saturation = compute_mean_hsv(image)

            # Compute standard deviation HSV
            std_hue, std_saturation, std_value = compute_std_hsv(image)

            # Compute Haralick texture features
            texture_features = compute_haralick_texture(image)
```

```
            # Create a dictionary for the feature vector with standardized names
            feature_vector = {
                'std_red': std_red,
                'std_green': std_green,
                'std_blue': std_blue,
                'mean_hue': mean_hue,
                'mean_saturation': mean_saturation,
                'std_hue': std_hue,
                'std_saturation': std_saturation,
                'std_value': std_value,
                'avg_contrast': texture_features[0],
                'avg_dissimilarity': texture_features[1],
            }

            # Ensure the feature vector matches the expected features
            ordered_feature_vector = [feature_vector[feature] for feature in
expected_features]
            feature_vectors.append(ordered_feature_vector)

        except rasterio.errors.RasterioIOError as e:
            print(f"Error reading {tile_path}: {e}")
            error_files.append(tile_file)
            continue

    return feature_vectors, error_files

def process_all_tiles(base_dir, batch_size=5000):
    """
    Process all tiles in the given directory, batch-wise, to extract features and
predict land classes.

    Args:
        base_dir (str): Base directory containing the tile subdirectories.
        batch_size (int): Number of tiles to process per batch.
    """
    all_feature_vectors = []
    all_predictions = []
    all_tile_files = []
    all_error_files = []

    for subdir, _, files in os.walk(base_dir):
        tile_files = [f for f in files if f.endswith('.tif')]
        subfolder_name = os.path.basename(subdir)
        num_batches = len(tile_files) // batch_size + (1 if len(tile_files) %
batch_size != 0 else 0)

        for batch_idx in range(num_batches):
            start_idx = batch_idx * batch_size
            end_idx = start_idx + batch_size
            batch_files = tile_files[start_idx:end_idx]
```

```
            try:
                batch_feature_vectors, error_files = process_batch(batch_files,
subdir, subfolder_name)
                all_feature_vectors.extend(batch_feature_vectors)
                all_error_files.extend(error_files)

                # Convert list of feature vectors to DataFrame
                batch_features_df = pd.DataFrame(batch_feature_vectors,
columns=expected_features)

                # Apply the pre-trained model for class predictions
                X_test = batch_features_df.values  # Ensure the correct order of
features
                batch_predictions = model_SVM.predict(X_test)  # Class predictions
                all_predictions.extend(batch_predictions)
                all_tile_files.extend([(subfolder_name, tile_file) for tile_file in
batch_files])
                print(f'Processed batch {batch_idx + 1}/{num_batches} in {subdir}')

            except Exception as e:
                print(f"Error processing batch {batch_idx + 1} in {subdir}: {e}")
                continue

    # Ensure lengths match before saving
    if len(all_tile_files) != len(all_predictions):
        print("Mismatch in lengths between tile files and predictions.")
    else:
        # Save the predictions with tile and subfolder information
        tiles_info = pd.DataFrame({
            'Subfolder': [t[0] for t in all_tile_files],
            'Tile': [t[1] for t in all_tile_files],
            'Prediction': all_predictions
        })

        output_excel =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/Model_Output/SVM_LC/Til
e_14_predictions_0108_v2.xlsx'
        tiles_info.to_excel(output_excel, index=False)
        print("Tile predictions saved to Excel file.")

    # Save the list of error files for review
    error_log_path =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/Model_Output/error_log.
txt'
    with open(error_log_path, 'w') as error_log:
        for error_file in all_error_files:
            error_log.write(f"{error_file}\n")
    print("Error log saved.")

# Example usage
```

```
base_dir =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/Segmentation/SegTest_Split_14x14'
process_all_tiles(base_dir)
```

Code 9: Aggregating tiles with predictions back together

- Prediction given to each .tif file and stored within the 5th band of the tile
- Tiles then aggregated back together to form a final output map
- File Name: *Aggregate_Tiles.py*

```
import os
import numpy as np
import rasterio
from rasterio.merge import merge
import pandas as pd

"""
This script performs the following tasks:
1. Loads a CSV file containing predictions for image tiles.
2. Maps each prediction label to a corresponding class number.
3. Reads image tiles, appends a new band representing the class number, and saves the modified tiles.
4. Processes the tiles in batches to manage memory usage effectively.
5. Merges the modified tiles into intermediate mosaic images.
6. Merges the intermediate mosaics into a final mosaic image.
"""

# Load the CSV file
csv_path =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/SubsetDroneforModelApp/Riverdrone/RiverPredictions3.csv'
predictions_df = pd.read_csv(csv_path)

# Print column names to verify
print("Columns in the CSV file:", predictions_df.columns)

# Define a mapping for each label to a class number
label_to_class = {
    'Water': 1,
    'Vegetation': 2,
    'Earth': 3,
    'Road': 4,
    'Building': 5,
    'Plastic': 6,
}

# Root directory containing subdirectories with TIFF tiles
```

```
root_tiles_dir =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/SubsetDroneforModelApp/
Riverdrone/RiverDroneTiles14'
# Output directory for modified tiles
output_dir =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/RiverDroneTiles14/River
dronemodified'
os.makedirs(output_dir, exist_ok=True)

# Batch processing parameters
batch_size = 5000  # Adjust batch size to balance memory and storage usage

def process_batch(batch_df, subfolder_path, subfolder_name):
    """
    Processes a batch of image tiles by appending a new band representing the class
number.

    Args:
        batch_df (pd.DataFrame): DataFrame containing the tile names and
predictions for the current batch.
        subfolder_path (str): Path to the subfolder containing the tiles.
        subfolder_name (str): Name of the subfolder (used for naming modified
files).

    Returns:
        list: List of paths to the modified tiles.
    """
    modified_files = []
    for _, row in batch_df.iterrows():
        tile_name = row['file_name']
        land_class = row['prediction']

        # Get the corresponding class number for the land class
        class_value = label_to_class.get(land_class, 0)  # Default to 0 if not
found

        tile_path = os.path.join(subfolder_path, tile_name)

        # Debugging statement to check the constructed path
        print(f"Processing tile: {tile_path}")

        if not os.path.exists(tile_path):
            print(f"Tile not found: {tile_path}")
            continue

        with rasterio.open(tile_path) as src:
            image = src.read()  # Read all bands

            # Ensure the image has 4 bands (RGBA)
            if image.shape[0] != 4:
```

```
                raise ValueError(f"Expected 4 bands (RGBA) in the image, but got
{image.shape[0]}")

            # Create a new band for the land class
            class_band = np.full((image.shape[1], image.shape[2]), class_value,
dtype=image.dtype)

            # Stack the original bands with the new class band
            modified_image = np.vstack((image, class_band[np.newaxis, ...]))

            # Ensure the modified image has exactly 5 bands
            if modified_image.shape[0] != 5:
                raise ValueError(f"Expected 5 bands in the modified image, but got
{modified_image.shape[0]}")

            # Create the output file path with subfolder name to ensure uniqueness
            modified_tile_path = os.path.join(output_dir,
f"{subfolder_name}_{tile_name}")

            # Write the modified image to a new file with compression
            meta = src.meta.copy()
            meta.update({
                'driver': 'GTiff',
                'height': modified_image.shape[1],
                'width': modified_image.shape[2],
                'count': 5,  # Number of bands (RGBA + class)
                'dtype': modified_image.dtype,
                'compress': 'lzw'  # Use LZW compression
            })

            with rasterio.open(modified_tile_path, 'w', **meta) as dst:
                dst.write(modified_image)

            modified_files.append(modified_tile_path)

    return modified_files

def merge_tiles(tile_paths, out_path):
    """
    Merges a list of image tiles into a single mosaic image.

    Args:
        tile_paths (list): List of paths to the image tiles to be merged.
        out_path (str): Path to save the resulting mosaic image.
    """
    # Open the tile files
    sources = [rasterio.open(tile_path) for tile_path in tile_paths]

    # Merge the tiles
    mosaic, out_trans = merge(sources)
```

```
    # Create output metadata
    out_meta = sources[0].meta.copy()
    out_meta.update({
        "driver": "GTiff",
        "height": mosaic.shape[1],
        "width": mosaic.shape[2],
        "transform": out_trans,
        "count": 5,  # RGB + Alpha + class band
        'compress': 'lzw'  # Use LZW compression
    })

    # Write the mosaic to disk
    with rasterio.open(out_path, "w", **out_meta) as dest:
        dest.write(mosaic)

    # Close all sources
    for src in sources:
        src.close()

    # Delete intermediate files to save space
    for tile_path in tile_paths:
        if os.path.exists(tile_path):
            os.remove(tile_path)
        else:
            print(f"File not found: {tile_path}")

def process_subfolders(predictions_df):
    """
    Processes all subfolders containing image tiles, modifies the tiles, and
creates intermediate mosaic images.

    Args:
        predictions_df (pd.DataFrame): DataFrame containing the tile names and
predictions.

    Returns:
        list: List of paths to the intermediate mosaic images.
    """
    subfolder_paths = [os.path.join(root_tiles_dir, subfolder) for subfolder in
os.listdir(root_tiles_dir) if os.path.isdir(os.path.join(root_tiles_dir,
subfolder))]
    intermediate_mosaics = []

    total_batches = sum(len(predictions_df[predictions_df['subfolder'] ==
os.path.basename(subfolder_path)]) // batch_size + 1 for subfolder_path in
subfolder_paths)
    processed_batches = 0

    for subfolder_path in subfolder_paths:
        subfolder_name = os.path.basename(subfolder_path)
        tile_files = [f for f in os.listdir(subfolder_path) if f.endswith('.tif')]
```

```
        for start in range(0, len(tile_files), batch_size):
            end = min(start + batch_size, len(tile_files))
            batch_df = predictions_df[(predictions_df['subfolder'] ==
subfolder_name) & (predictions_df['file_name'].isin(tile_files[start:end]))]

            print(f"Processing batch in {subfolder_name} from {start} to {end}")
            modified_files = process_batch(batch_df, subfolder_path,
subfolder_name)

            # Merge the batch of modified tiles
            batch_output_path = os.path.join(output_dir,
f"{subfolder_name}_mosaic_{start}_{end}.tif")
            merge_tiles(modified_files, batch_output_path)

            intermediate_mosaics.append(batch_output_path)

            processed_batches += 1
            progress_percentage = (processed_batches / total_batches) * 100
            print(f"Progress: {progress_percentage:.2f}%")

    return intermediate_mosaics

# Process the entire CSV file
intermediate_mosaics = process_subfolders(predictions_df)

# Merge all intermediate mosaics into the final mosaic
final_output_path =
'/Users/mianikitacroft/Documents/Edinburgh/Uni/Dissertation/Model_Output/Riverdrone
_stitchedoutput.tif'
merge_tiles(intermediate_mosaics, final_output_path)

print(f"Final mosaic saved to {final_output_path}")
```