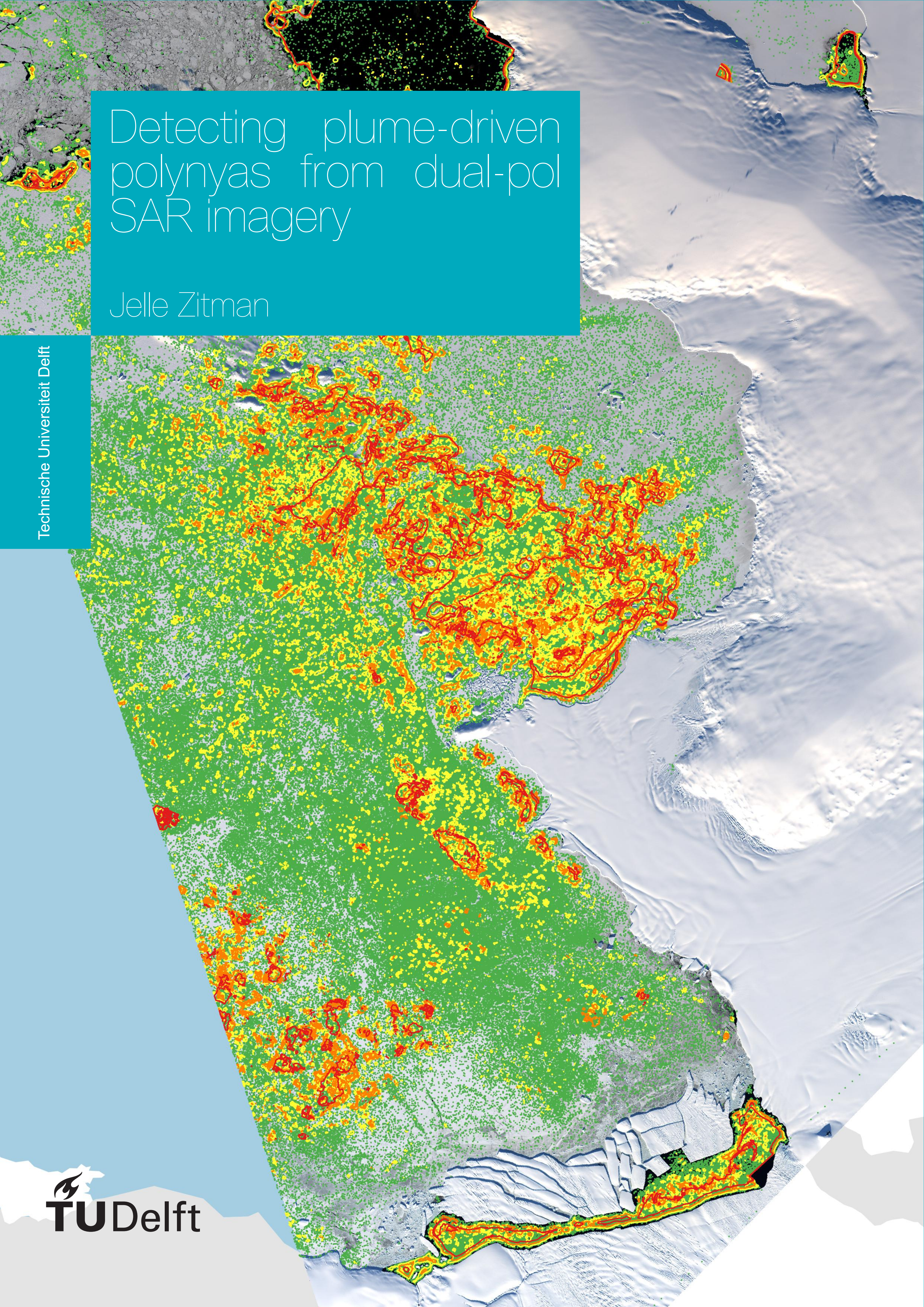
Detecting plume-driven polynyas from dual-pol SAR imagery

Jelle Zitman

Technische Universiteit Delft



Detecting plume-driven polynyas from dual-pol SAR imagery

by

Jelle Zitman

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on August, 25, 2022 at 14:30.

Student number:	4376048	
Project duration:	September 1, 2021 – August 25, 2022	
Thesis committee:	Dr. Ir. S. I. M. Lhermitte,	TU Delft, daily supervisor
	Dr. Ir. B. Wouters,	TU Delft, supervisor
	Ir. M. Izeboud,	TU Delft, supervisor

TU Delft

Abstract

Antarctic ocean temperatures are rising due to climate change, causing land ice to melt at increasingly higher rates. Ice shelf bottom melt is a key factor responsible for Antarctic ice mass loss and as such understanding melt processes in the Antarctic is therefore key to more accurately predict how the global sea level will respond to climate change in the foreseeable future. Basal melt results in the formation of both basal melt channels underneath an ice shelf and persistent sea ice wakes (named plume-driven polynyas) at the ice shelf shoreline. The goal of this research is to develop a method that can help to automatically infer basal melt locations along the Antarctic shoreline with significantly increased spatio-temporal resolution compared to previously researched basal melt detection methods. We infer basal melt locations by detecting plume-driven polynyas. We used dual-pol (HH/HV) Sentinel-1 EW SAR data (40x40m resolution) in combination with GLCM textural features as input for a random forest classification that differentiates images as water or ice in four sub-classes: undisturbed 'open' water, disturbed 'rough' water, sea ice and (floating) land ice. We assessed what the advantages and limitations of this approach were for plume-driven polynya detection by performing water-ice (sub-class) classifications and examining which GLCM features proved most useful, what GLCM window size is preferred, and how classification can be aided by post-processing classified images. We computed GLCM textures for window sizes *w* = [5,11,21] and created a classifier for each choice (`GLCM5`, `GLCM11` and `GLC21`) and compared results to a classifier based on original dual-pol SAR data (`BASE`). Via cross validated recursive feature elimination we determined that 'sum average' (HH and HV polarization) and 'difference variance' (HV polarization) were most useful for separation of water and ice classes (`HH_savg`, `HV_savg` and `HV_dvar`). Our results have shown that using GLCM texture based dual-pol classifiers improves water-ice classification significantly compared to dual-pol only classifiers, although using `HH_savg` and `HV_savg` instead of orignal dual-pol data comes at a cost of reduced spatial resolution. Water-ice classification accuracy of `BASE` was 92.2% (kappa = 84.4%) was increased to 95.9% (kappa = 91.5%) for `GLCM5`, 96.3% (kappa = 92.7%) for `GLCM11` and to 96.5% (kappa = 93.0%) for `GLCM21`. From a spatial context, `GLCM21` showed an insufficient ability to detect small-scaled bodies of water at a sub-kilometer scale. `GLCM5` showed unsatisfactory results in terms of sea ice classification. `GLCM11` showed highest robustness in both these performance aspects and proved to be most successful classifier for the application of polynya detection. Using an area filter as a post-processing step proved successful when a classifier is based on GLCM data with a window size no larger than *w*=11. Noise output (small regions of falsely classified open water pixels) was heavily reduced via this form of post-processing and significantly increased polynya detection performance. The final classified product however still contained too many incorrectly classified water regions of similar spatial scales as plume-driven polynyas to be able to apply this algorithm as a reliable automated polynya detection method. We urge to build upon this SAR-based detection method, by using additional non-GLCM input features or using extra post-processing steps, such as temporally filtering water body presence, until results are satisfactory for a fully automated plume-driven polynya detection algorithm. The method presented here has the potential to make detection significantly faster, easier and more accessible than the current methods available. Lastly, in its current state, this method can already be used to validate predicted locations of basal melt by ocean-ice sheet models and DEM-based methods.

Preface

This year has been extremely educational and challenging, but, for a multitude of reasons, ultimately very tough for me. I'd therefore like to take this opportunity to express my gratitude for all the people that have helped me throughout the year.

First of all, a big thank you to my supervisors. Stef, it has been a joy to work together. It has been a chaotic year in many aspects, but I always found you a very kind person to work with, which I value greatly. Your advise was always spot on and I have learned a lot from your way of working. We may unfortunately have not seen each other as much as is normally the case, but the (online) meetings we've had were always helpful. Bert, thank you for being available as the final supervisor in relatively short notice. Your feedback was honest and I highly appreciate the time that you've put in for the supervision. Finally, a massive thanks to Maaike. I'm extremely grateful for all the personal contact that we've had, especially during the second half of the past year. You were always available when I asked to meet, you always gave quick and thorough feedback on intermediate results, and were glad to answer the many questions I threw at you. Your help has been one of the reasons why I've been able to work as hard as I have done over the last couple of months and I thank you immensely for it. Thank you all for your time, effort and honesty. I appreciate it greatly and I wish you all the best.

Secondly, I have to thank my family! Mom, dad, thank you so much for your continuous support throughout the past 8 years. Your support has made me the person I am today. Dad, I want to thank you personally for the help you offered. Even though I in the end haven't made much use of it I've always kept it in the back of my mind. Mom, thanks for the conversations we've had about graduating. I can express whatever I want to you and our conversation in May is one of the reasons my work went much better from then on. Camee, the moments we've had talking about my thesis were sparse given our different interests in our studies, but you've helped me a great deal with the mental side of things. Know that I'll help you in any way I can for the continuation of your own studies, you only need to ask. Thanks, I love you all.

Lastly, a HUGE thanks to all my friends! Your support has helped me tremendously. Derek, thanks for an amazing time as room mates. Some moments were really tough this year and without asking you helped me and lifted my spirits. I cannot ask for a better friend, thank you. Sjonnie, we've had similar struggles this year and I appreciate the talks we've had on how to deal with it. You helped me a great deal by supporting my approach and I hope I helped you as well in a similar fashion. Koen, Nienke and Romy, thanks to you as well for having my back and letting me work at your place when I wanted to. Iris, thank you a great deal for listening to me when we talked about this project. I needed it and you were there for me. Paula, Luuk, Tim, thanks for an amazing adventure in the Swedish mountains. Kim, Mitchel, Niek, thanks for working together during the end of the year and during the summer months. I furthermore have to thank you as Soepel for the amazing holiday we had together in the Austrian Alps. I hopped aboard the train pretty exhausted, but you guys fixed me. Lastly, my biggest thanks has to go to Martine. I'm not kidding when I say I could have not have achieved what I have done the past year without you. Your continuous support, your personal interest, working together at IDE or at home, I could always count on you. Thank you so much.

Thanks as well to everyone who helped me that I've not mentioned personally, I love the backing you gave me. It feels amazing to have this many special people around me.

Jelle Zitman
Utrecht, August 18, 2022

Contents

List of Figures

List of Tables

1
Introduction

Ocean temperatures are rising due to climate change [IPCC, 2014]. Warmer ocean temperatures have great effects on Arctic and Antarctic environments, where land ice is melting at increasingly higher rates [e.g. IPCC, 2014; Jourdain et al., 2017; Alley et al., 2019]. Antarctic ice mass loss accelerates global sea level rise [IPCC, 2014], but its contribution is still one of the least understood contributing factors [Pattyn and Morlighem, 2020; IPCC, 2014]. Understanding melt processes in the Antarctic is therefore key to more accurately predict how global sea level will respond to climate change in the foreseeable future.

1.1. Basal Melt of Antarctic Ice Shelves

Most mass loss of the Antarctic Ice Sheet is due to melt of Antarctic Ice Shelves. Many factors are responsible for ice shelf melt, but bottom melt of an ice shelf due to warm ocean currents is one of its most important drivers. Research by Alley et al. [2019]; Lazeroms et al. [2018]; Dutrieux et al. [2014]; Rignot and Steffen [2008]; Lhermitte et al. [2020] has shown that it significantly affects the stability of ice shelves and is responsible for accelerations in sea level rise.

Basal melt occurs when warmer circulatory ocean layers, e.g. Circumpolar Deep Water (CDW), flow upwards along the bottom of ice shelves [Brocq et al., 2013; Alley et al., 2016] and can be responsible for creating 1-10km wide trenches in the ice [Sergienko, 2013; Gourmelen et al., 2017; Alley et al., 2019]. These trenches usually form along the ice shelf's outer edges, though such trenches have also been observed along ice shelf center lines as well. This is the case for one of the west Antarctic ice sheet's glaciers, Pine Island Glacier [Mankoff et al., 2012; Alley et al., 2019]. This glacier shows clear signs of calving due to basal melt, shown in Figure 1.1 for reference.

1.2. Plume-driven Polynyas

As Figure 1.1 shows, apart from melting the bottom of an ice shelf, warmer oceanic waters also melt part of the sea ice where this current reaches the ocean surface. These sea ice wakes are usually confined to the shoreline of the ice shelf at the end of a basal melt channel as they are caused by buoyant meltwater plumes [Lazeroms et al., 2018]. In this study we call these wakes *plume-driven* polynyas. *Polynya* is the preferred nomenclature for regions of open water or thin ice, found in polar sea ice zones where thicker sea ice would be expected based on climatological conditions [Barber and Massom, 2007]. Polynyas can form due to various external factors, but generally fall in two categories: polynyas formed by mechanical forces such as wind, or polynyas formed by convective forces such as warm buoyant water plumes [Williams et al., 2007]. Wind-driven polynyas are generally very large in size and can span up to thousands of kilometers [Barber and Massom, 2007]. Their thermal signature is low, as such water bodies are only ice-free due to harsh meteorological conditions, and can show seasonal variability if climatological conditions show large seasonal variations. On the other hand, plume-driven polynyas are characterized by strong thermal signatures due to warm buoyant water plumes [e.g. Alley et al., 2019; Hellmer et al., 2012] being the cause of melt. These polynyas therefore do not freeze easily if the ocean currents do not vary seasonally. Furthermore, plume-driven polynyas

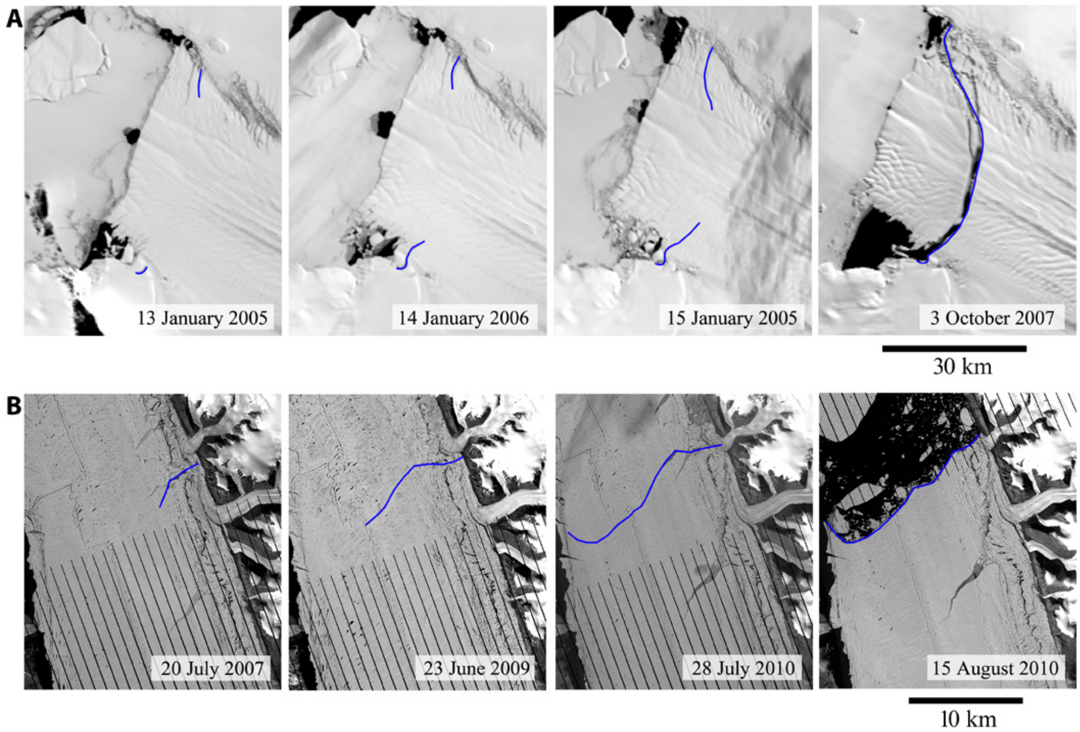


Figure 1.1: Optical imagery of Pine Island Glacier, showing calving fronts due to basal melt in blue. At the end of basal channels, adjacent to the ice shelf shoreline, plume-driven polynyas are visible in the sea ice.
Courtesy of Alley et al. [2019].

are considerably smaller in size than wind polynyas, having diameters roughly in the same order as the width of basal melt channels [e.g. Alley et al., 2019].

Research has been done to predict where basal melt is occurring. Gladish et al. [2012]; Lazeroms et al. [2018]; Sergienko [2013], among others, have modelled basal melt distribution and influence on ice shelf stability using ocean models. Others, such as Dutrieux et al. [2014] have used in-situ measurements to describe basal melt characteristics. Another means is to use satellite imagery in combination with an ice sheet model to look at ice shelf dynamics directly, performed by Lhermitte et al. [2020], or to combine satellite imagery with a Digital Elevation Model (DEM) to detect basal melt pathways [Shean et al., 2019]. Consequently, research into basal melt via different methods aids in increasing the rate at which we understand Antarctic basal melt, and it allows us to draw more accurate conclusions on its consequences. However, these methods have their own drawbacks. Ice sheet models are computationally expensive, in-situ observations are often sparse and not easily obtained, and the use of DEMs results in a low temporal resolution of basal melt predictions. It would therefore be beneficial to use satellite data to (indirectly) study basal melt, such that the temporal and spatial resolution of predictions remains reasonably high.

In this light, we propose a new method to research Antarctic basal melt. We do not detect basal melt directly, but use remote sensing to detect plume-driven polynyas, which occur as a direct consequence of basal melt channels. The goal of this method is to develop a method to infer basal melt locations along the Antarctic shoreline with significantly increased spatio-temporal resolution compared to previously researched basal melt detection methods.

1.3. Synthetic Aperture Radar

A variety of satellite data could be used for polynya detection. Ideally, we use data that are independent on weather conditions, sun elevation (due to long periods of polar winter), have a spatial resolution higher than 1km, and a high enough temporal resolution to revisit locations multiple times per year. For this purpose, C-band Synthetic Aperture Radar (SAR) data is well suited [e.g. Dierking, 2010; Karvonen and Hallikainen, 2009]. C-band SAR is an active sensor that sends radar waves, which penetrate clouds, to measure surface characteristics, meaning it can operate during day and night time [Xu and Li, 2015; Karvonen and Hallikainen, 2009].

1.3.1. Polarization

Depending on the type of sensor SAR data is polarized in different ways, meaning that the plane in which radar waves and the surface interact is defined by the sensor. Each sensor can pick up different surface characteristics as surface depolarization is dependent on surface roughness and on surface composition [Mermoz et al., 2009]. Therefore, SAR data is able to reveal surface characteristics in different ways, depending on the sensor. The polarization of SAR data can be VV (vertically transmitted, vertically received), VH (vertically transmitted, horizontally received), HH (horizontally transmitted, horizontally received) or HV (horizontally transmitted, vertically received). VV and HH are co-polarization backscatter intensities, whereas VH and HV are cross-polarization backscatter intensities. SAR data can be single-polarized (single-pol) by measuring VV or HH or dual-polarized (dual-pol) by measuring VV and VH or HH and HV.

1.3.2. Surface Backscatter Characteristics

Though SAR has many advantages on an operational level, detecting water bodies and sea ice from it is challenging [Karvonen et al., 2005]. Backscatter intensities of water and sea ice have very similar ranges and values are very much dependent on local wind speeds, wind directions and the incidence angle of the sensor. Wind has an immediate effect on the sea state. In very calm conditions, water is a highly specular surface and acts as a mirror, leading to low co-polarization backscatter intensities (≤-30 [dB]) at an incidence angle of 30°. Conversely, when a sea state is rougher due to higher wind speeds, ripple waves or large waves can significantly increase the backscatter intensity of the water body [Karvonen et al., 2005] ($\simeq$ -20 [dB]). HH is a useful polarization as it is not very sensitive to wind roughness [Long et al., 1996], which is helpful to distinguish smooth sea ice from water in windy conditions. The range of backscatter intensity of sea ice is large as well. Sea ice can be very smooth, thereby mimicking calmer open water conditions in terms of backscatter intensity. However, co-polarization backscatter intensities of smooth or wet sea ice are still higher than very calm open bodies of water, due to (more) volumetric scattering and higher surface roughness, but intensities can be comparable to rougher water conditions ($\simeq$ -20 [dB]). In the case that sea ice is ridged, or when sea ice is covered in a thick snow layer, surface roughness increases significantly. Furthermore, radar waves are able to penetrate such surfaces more easily due to higher porosity of ice and snow compared to water and are scattered volumetrically, leading to much higher backscatter intensities ($\geq$ -10 [dB]). Water and ice surfaces have somewhat different characteristics for cross-polarization data. Calm, flat water bodies still show very low backscatter intensities ($\leq$ -30 [dB]) as surface roughness and volumetric scattering are minimal. For rougher sea states, the backscatter intensities are more similar to those of calm water ($\simeq$ -30 [dB]), as reflection is mostly due to surface scattering, so the incoming radar signal experiences little depolarization [Freeman and Durden, 1998]. For smooth sea ice surfaces, a significantly larger portion of the incoming radar waves is scattered volumetrically, leading to higher cross-polarization backscatter intensities ($\simeq$ -25 [dB]) than rough water. In this way, cross-polarized data is helpful for the distinction of sea ice and water, as subtle differences between rougher water and smooth ice are better distinguishable [de Roda Husman et al., 2021], albeit at the expense of a poorer signal to noise ratio than co-polarization data [ESA, 2012]. Cross-polarization backscatter intensities for older ridged ice and snow covered ice remain high as volumetric scattering is dominant for this surface type and as surface roughness is large as well.

1.3.3. Textural Features

Overall, the use of SAR data to classify water and ice has a lot of potential, despite the inherent weaknesses related to similar surface characteristics of rougher waters and smoother ice. Research done by Karvonen and Hallikainen [2009]; de Roda Husman et al. [2021]; Dierking [2013] has proven that SAR can be well-suited for ice type classification. For ice-water classification, Karvonen et al. [2005]; Lohse et al. [2021]; de Roda Husman et al. [2021] showed that dual-pol SAR data in particular can be useful, but that additional textural information is needed for more accurate classification. Spatial information can be derived from co- and cross-polarization backscatter intensities by computing statistical texture metrics around each pixel of a SAR image. This is called a Gray Level Co-occurence Matrix (GLCM) and is a tabulation of different combinations of grey levels on a specified area surrounding any pixel in the input image [Haralick et al., 1973; Conners et al., 1984]. The extent of this area is determined by a prescribed window size *w*. Research done by de Roda Husman et al. [2021], has shown the advantageous use of GLCM textures in the context of river ice classification. Other studies, such as Lohse et al. [2021], have used a similar approach for sea ice detection and concluded that specific texture metrics can be useful for sea ice classification. This is relevant for plume-driven polynya detection, as there is a need to distinguish any ice type from bodies of water as accurately as possible. For one, Lohse et al. [2021] have shown that window size influences classification accuracy. Generally, larger windows are recommended for sea ice state detection, although this comes at a price of spatial resolution. We are therefore interested in the (dis)advantages of dual-pol SAR data in combination with GLCM texture features to detect plume-driven polynyas.

1.4. Study Goal

As basal melt and its extent under Antarctic Ice Shelves is still not fully understood, it is valuable to analyze its dynamics. Basal melt observations are still sparse and difficult to acquire and detecting plume-driven polynyas can give more insight in the presence of basal melt channels. Using SAR satellite data can be a valuable asset in detecting such polynyas as the spatio-temporal coverage of SAR data is significantly higher than in-situ measurements and DEMs. Differentiating water and ice from SAR data can be challenging as both surface types show similarities in cross- and co-polarized SAR images. Furthermore, the use of GLCM textural features has been proven effective in improving water-ice classification accuracy. Studies so far have used SAR data to detect sea ice types and general open water bodies, instead of plume-driven polynyas in particular and have mostly focused on local regions. With this in mind, we focus on an approach to automatically detect plume-driven polynyas, following the research question below:

What are the strengths and limitations of classifying and post-processing dual-pol SAR data and GLCM features for the detection of plume-driven polynyas?

To answer this question, we ask the following sub-questions:

1. *How is open water classification affected when basing a classifier on dual-pol GLCM features, compared to classification where only dual-pol SAR data is used?*
   (a) *Which GLCM features based on dual-pol polarized SAR data prove most useful for water detection?*
   (b) *How does the GLCM window size affect classification performance of water bodies?*
2. *In what way do post-processing steps improve plume-driven polynya detection from ice-water classified images?*

1.5. Approach

1.5.1. Google Earth Engine

A disadvantage of using satellite data on continent-wide scales is that data sets become extremely large. Big data analysis requires large computational storage capacity and calculations can quickly become computationally expensive. For this problem, the Google Earth Engine (GEE) provides a solution. The GEE is a cloud-based earth observation platform that allows any user to perform geospatial analyses on a planetary scale through cloud computing [Gorelick et al., 2017]. The GEE comes with a large variety of pre-processed data sets ready for use and is well-suited to analyze Antarctic climate change dynamics. For exactly this reason the GEE is a useful tool for Antarctic polynya detection and will be used as computational platform in this study.

1.5.2. Classification

Many classification algorithms are available, each with their own strengths and shortcomings. First of all, one can choose to use unsupervised or supervised classification. Via unsupervised classification, an image will be classified in a prescribed number of classes, but this method does not provide a means to assign labels to each class beforehand. This means that images have to be analysed using expert knowledge by comparing it to reference data to interpret the classification. Supervised classification does provide the option to classify the SAR image in prescribed classes, by training data beforehand. No extra step is needed after classification is done and thus supervised classification is preferred, as we want to automate the detection process.

1.5.3. Thesis Outline

To study what GLCM features affect the classification of open water bodies, we developed and analysed multiple classifiers. To analyse the detection accuracy of plume-driven polynyas, we have post-processed the classified images. In chapter 2 we describe which data sets were used, how they were pre-processed and filtered, and define a study region. In chapter 3 we show what textural features were selected and which supervised Machine-Learning (ML) algorithm is used for classification. Furthermore, we show how we have chosen GLCM window sizes and which classifiers we have created and from the selected textural features, ML algorithm and window size. Moreover, we lay out how each classifier was tested on their performance. In chapter 4 we then present classification results and assess how different classifiers compare to each other. Furthermore we apply post-processing steps to the classified data set. Moving to chapter 5, we discuss to what ends the classifier can be used, whether post-processing steps aid in the detection of plume-driven polynyas and provide recommendations for future studies regarding this topic. Lastly, we conclude on the efficacy of adding textural features to dual-pol SAR data for open water classification and the use of post-processing steps on classified images for plume-driven polynya detection in chapter 6.

2

Satellite Imagery & Study Area

This study uses SAR imagery to detect open water bodies, after which plume-driven polynyas are distinguished from other bodies of water in post-processing. As we perform a supervised classification, optical reference data is needed to first manually prescribe feature data to predefined classes. We use data available on the GEE and define a study region based on their spatio-temporal distribution. The presence of optical imagery determines which SAR images are suited as training or validation images. We apply several pre-processing and filtering steps to create this combined data set.

2.1. Optical Reference Imagery

For optical imagery we use LandSat 8 (L8) scenes, which are open-source available. Specifically, we use raw images from Collection 1, Tier 2, as these are available over the Antarctic continent and adjacent waters. Tier 2 raw images display at-sensor radiance that did not meet the quality requirements of Tier 1 images. Factors that determine this quality check are, among others, significant cloud coverage and a lack of ground control. This lack in quality is overcome by manually selecting images of the highest quality during scene selection.

2.1.1. Quality Filter

We have selected optical images based on the following criteria: 1) the footprint of the image should intersect with the coast line of any Antarctic Ice Shelf, 2) the cloud coverage in the image should be lower than 20%, and, to discard images that are too dark, 3) the image should have a maximum brightness above a predetermined threshold. The selection of optical images that we obtained from these filtering steps was matched with SAR images and separated into training and validation images.

2.2. Synthetic Aperture Radar Imagery

When wanting to use SAR data, the GEE provides access to ESA's Sentinel-1 (S1) SAR Level-1 products. The Level-1 product is acquired from a dual-pol C-band SAR sensor [ESA, 2012] and is calibrated and ortho-corrected. Along the Antarctic coast the sensor in operation is mostly single-pol HH, but dual-pol HH and HV is also available. For this reason we use C-band SAR data and we perform our analysis on HH and HV polarized backscatter intensities. The data have resolutions specific to the type of sensor used for acquisition. An advantage of working in the GEE is that most of the other usual pre-processing steps (e.g. SLC processing, radiometric calibration, terrain correction and thermal noise removal) are already accounted for using the Sentinel-1 Toolbox [ESA, 2012].

2.2.1. Pre-Processing

For our study, we used scenes in HH and HV polarization acquired from the Extra Wide Swath instrument mode (EW). These scenes have a footprint of 400x400km and a spatial resolution of 40x40m [ESA, 2012]. Next to backscatter intensity in HH and HV polarization, Sentinel-1 images come with an incidence angle band, whose values range from 20 to 47 degrees [ESA, 2012]. We pre-processed both image bands by applying an angle correction presented by Topouzelis and Singha [2016] on all images to normalize HH and HV backscatter intensities to an effective angle of (roughly) 30 degrees.

2.2.2. Reference Data Filter

For the selection of training and validation images, we filtered S1 images acquired from 2014-10-01 to 2022-04-01 in several criteria. 1) The footprint of a scene should intersect with the shoreline of any ice shelf, 2) the scene should contain both HH and HV polarized data, and 3) the scene should intersect with the footprint of an optical image from the already filtered L8 scenes, whilst having been acquired within 4 hours of each other. A spatial overview of amount of S1-L8 matches along the Antarctic coast is presented in Figure 2.1 for reference.

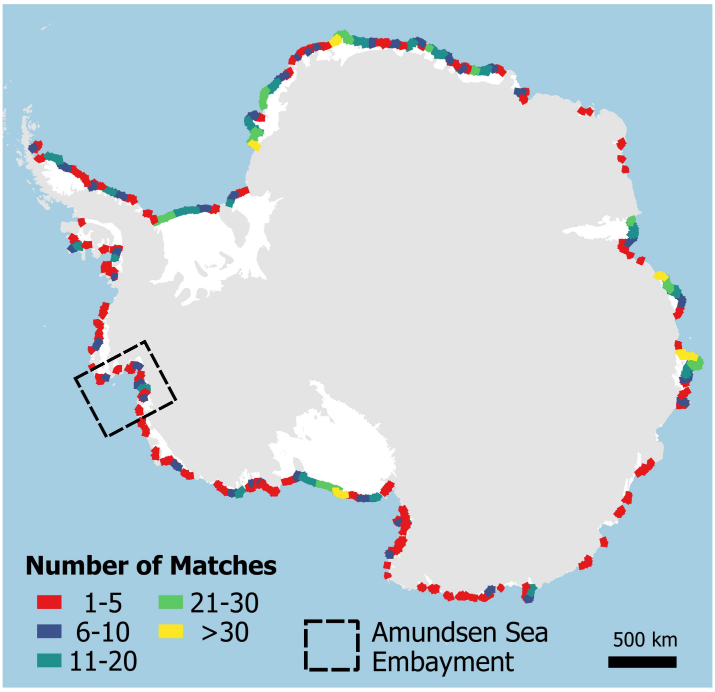


Figure 2.1: Number of dual-pol S1 images acquired within 4 hours of a L8 scene.

2.3. Study Region

Based on Figure 2.1 and on prior knowledge of the presence of plume-driven polynyas [Bindschadler et al., 2011; Alley et al., 2019; Maqueda et al., 2004; Stewart et al., 2019], we chose a study region for classification. Observations from optical imagery [e.g. Alley et al., 2019] have shown that most plume-driven polynyas are present in the Amundsen Sea Embayment and as such we choose this coastal region as our region of interest. Although in this region does not contain most S1-L8 matches, enough matches ($\sim$ 20) are available to create training and validation data from. We extend the study area some distance west-ward to include the coast near Coney Island. We do this to increase the number of available images, and to make sure that the training and validation data are not acquired in one secluded region of the Antarctic Coast. In this manner, we aim make the classifier more robust and applicable along the entire coast, without the need to resample training data. We filtered the match data set on the study region and performed a final visual quality check where we discarded S1 images that had too little overlap with their L8 match. In the end, the S1 image collection used for this study consists of 13 images, spanning from 2018-2021, and contain at least one L8 match each. We have separated this collection in a training collection of 9 images and a validation collection of 4 images to ensure that the ratio of number of training samples to number of validation samples is roughly 0.7. An overview of the training and validation data, including useful metadata, is given in Table 2.1.

Table 2.1: Metadata of Sentinel-1 and LandSat-8 images used as training and validation images. Images are index based on the coordinates of each Sentinel-1 image's footprint. Image IDs of each S1 and L8 image are given in Table A.1.

Match ID	Data Set	Satellite	Local Date	Local Time	Types of Water Bodies in Image
T1	Training	Sentinel-1	2017/12/26	20:52:24	Open Sea, Water Leads, Wind Polynya, Plume Polynya
		LandSat-8	2017/12/26	23:28:03	
		LandSat-8	2017/12/26	23:28:27	
T2	Training	Sentinel-1	2017/12/21	20:44:13	Open Sea
		LandSat-8	2017/12/22	00:09:32	
T3	Training	Sentinel-1	2018/12/18	20:28:00	Open Sea
		LandSat-8	2018/12/18	23:46:17	
T4	Training	Sentinel-1	2021/12/26	20:28:17	Open Sea
		LandSat-8	2021/12/26	23:46:42	
T5	Training	Sentinel-1	2018/01/11	20:19:41	Wind Polynya, Plume Polynya
		LandSat-8	2018/01/11	23:27:57	
T6	Training	Sentinel-1	2018/12/13	20:19:48	Open Sea, Water Leads, Wind Polynya
		LandSat-8	2018/12/13	23:28:08	
T7	Training	Sentinel-1	2017/12/13	21:28:20	Open Sea
		LandSat-8	2017/12/13	23:37:50	
T8	Training	Sentinel-1	2018/01/07	20:31:08	Open Sea
		LandSat-8	2018/01/07	23:31:35	
T9	Training	Sentinel-1	2018/01/08	20:33:39	Open Sea, Wind Polynya
		LandSat-8	2018/01/08	23:36:21	
V1	Validation	Sentinel-1	2017/12/19	21:00:36	Wind Polynya, Plume Polynya
		LandSat-8	2017/12/19	23:21:54	
V2	Validation	Sentinel-1	2017/12/23	20:27:53	Open Sea, Water Leads, Wind Polynya, Plume Polynya
		LandSat-8	2017/12/23	23:57:10	
		LandSat-8	2017/12/23	23:57:34	
V3	Validation	Sentinel-1	2020/01/06	20:28:05	Open Sea, Water Leads
		LandSat-8	2020/01/06	23:46:37	
V4	Validation	Sentinel-1	2017/12/18	20:58:33	Open Sea, Water Leads
		LandSat-8	2017/12/18	23:17:55	

2.4. GLCM Features

To introduce extra information to the original HH and HV polarized SAR data, we compute GLCM textural features. Through these computations, a prescribed amount of pixels around a center pixel are statistically interpreted in various ways, each resulting in a unique texture metric. In the GEE, 17 GLCM output features are computed per per input band, resulting in 36 unique input features (including HH and HV) for each classifier. As stated by Haralick et al. [1973], GLCM features are computed from entries in three types of gray-tone spatial-dependence matrices: $p(i,j)$, $p_{x+y}(k)$ or $p_{x-y}(k)$. $p(i,j)$ is the (i,j)th entry in the original normalized gray-scale matrix. Entries of the other adjusted gray-scale matrices are described via

$$p_{x+y}(k) = \sum_{\substack{i=1 \\ i+j=k}}^{N_g} \sum_{j=1}^{N_g} p(i,j), \quad k = 2, 3, \ldots, 2N_g \tag{2.1}$$

used for computing 'summed' features, and

$$p_{x-y}(k) = \sum_{\substack{i=1 \\ |i-j|=k}}^{N_g} \sum_{j=1}^{N_g} p(i,j), \quad k = 0, 1, \dots, N_g - 1 \tag{2.2}$$

which is the notation for computing 'difference' features. In both equations, N_g represents the number of distinct grey levels in the quantized image. 13 unique features are extracted by computing different statistics from each matrix. In a similar but more elaborate fashion, Conners et al. [1984], present six extra unique features, of which four are computed in the GEE. For clarity, we present an overview of names and descriptions of these 17 GLCM features in Table 2.2 .

Table 2.2: Names and abbreviations of GLCM features, originally described by Haralick et al. [1973] and Conners et al. [1984].

Index	Feature Name	Abbreviation	Matrix	Authors	Description
1	Angular Second Moment	ASM	$p(i,j)$	Haralick	Number of repeated grey-level pairs (energy)
2	Contrast	CONT	$p(i,j)$	Haralick	Local contrast of grey-levels
3	Correlation	CORR	$p(i,j)$	Haralick	Correlation between pairs of grey-levels
4	Variance	VAR	$p(i,j)$	Haralick	Spread in grey-levels
5	Inverse Difference Moment	IDM	$p(i,j)$	Haralick	Homogeneity of grey-levels
6	Sum Average	SAVG	$p_{x+y}(k)$	Haralick	Mean of 'summed' grey-level matrix
7	Sum Variance	SVAR	$p_{x+y}(k)$	Haralick	Variance of 'summed' grey-level matrix
8	Sum Entropy	SENT	$p_{x+y}(k)$	Haralick	Entropy of 'summed' grey-level matrix
9	Entropy	ENT	$p(i.j)$	Haralick	Randomness of grey-levels
10	Difference Variance	DVAR	$p_{x-y}(k)$	Haralick	Variance of 'difference' grey-level matrix
11	Difference Entropy	DENT	$p_{x-y}(k)$	Haralick	Entropy of 'difference' grey-level matrix
12	1st Correlation Measure	IMCORR1	$p(i,j)$	Haralick	Extra information measure of cluster correlation (1)
13	2nd Correlation Measure	IMCORR2	$p(i,j)$	Haralick	Extra information measure of cluster correlation (2)
14	Dissimilarity	DISS	$p(i,j)$	Conners	Distinction between grey-levels (named Local Homogeneity in Conners et al. [1984])
15	Inertia	INERTIA	$p(i,j)$	Conners	Periodicity of grey-levels
16	Cluster Shade	SHADE	$p(i,j)$	Conners	Uniformity of grey-levels
17	Cluster Prominence	PROM	$p(i,j)$	Conners	Proximity of grey-levels

For a full description of the formulas behind these features, we urge to consult Haralick et al. [1973] for features 1-13, and Conners et al. [1984] for features 14-17.

3
Method

In this study we compare different classifiers on their ability to separate open water from ice from dual-pol SAR imagery. We created multiple classifiers using different selections of input features. For this we computed texture features from backscatter intensity in HH and HV polarization for various window sizes. After classification, we applied post-processing steps to translate the classified open water bodies to polynyas. Every step in this process is discussed in detail below.

3.1. Data Sampling

We sampled data in different (sub-)classes to use for classification. We divided data in two main classes, *Water* and *Ice*, each consisting of two sub-classes: *Open Water*, *Rough Water*, *Sea Ice* and *Floating Land Ice* respectively. S1 images were manually assigned to a (sub-)class, based on several criteria from L8 images (`I`) and S1 HH (`II`) and HV (`III`) polarized backscatter intensities. Part of an S1 image is assigned as *Open Water* if `I` clearly shows a water body, while `II` and `III` are both near minimum intensity values. It is seen as *Rough Water* if `I` shows water, but either `II` or `III` has low to medium values, while the other has low to very low values. For *Sea Ice* the criteria of `II` and `III` are equal to that of *Rough Water*, but now `I` shows a presence of ice instead of water. Finally, part of the image is considered as *Floating Land Ice* when `I` shows ice and `II` and `III` both show high backscatter intensities.

Training and validation data were created by extracting data from 1,000 randomly selected pixels for every sub-class in each image. Sampling 1,000 pixels was an imposed computational upper limit. This resulted in a training data set consisting of 9,000 samples per feature per sub-class (i.e. 18,000 per class) and a validation data set of 4,000 samples per feature per sub-class (8,000 per class).

3.2. GLCM Textures

3.2.1. Feature Selection

As mentioned in chapter 1, spatial information is derived from HH and HV backscatter intensities by computing various GLCM features. GLCM texture calculations provide 17 output features per input band, given in Table 2.2. To analyse the added value of each feature (both for HH and HV), we performed a cross validated recursive feature elimination (RFECV). This method tests the accuracy of a classifier for a chosen number of features, ranging from 1 to all features available, on a subset of the training data set. In our case we divide the training data in three folds, and then compute the average accuracy over all folds. We can then apply a normal recursive feature elimination (RFE) on the full training data set to determine the *n* most important features. The algorithm determines the least important feature for each iteration and discards that feature until the desired number of features in order of highest importance is computed.

Apart from feature elimination, we also tested how statistically dependent training labels were from classified labels. We did this by performing a student t-test on each classification outcome. The statistical dependency is expressed in a p-value, where values lower than a threshold (usually 0.05) indicate that two variables are likely independent or unrelated. Conversely, higher p-values indicate that two variables are likely dependent or closely related.

We use the accuracy and p-values to assess which features are best for the classifier. We do this by first looking for the number of features where accuracy is significantly high, and the p-value has approached or reached its maximum. Then, we also looked at how much a feature correlates to any other feature in each data set. Features that were correlated more than 90% to each other were considered highly correlated and of these features only the feature with the highest importance score was used as input data for the classifier. We chose to do this to minimize the number of input features, so that we were able to more closely look at the influence that a feature has on the classification output. We have made the assumption that excluding highly correlated features does not impact the classification result significantly, and that for this reason such omissions are allowed. We discuss the eventual selected features in more detail in chapter 4.

3.2.2. Window Size

For the detection of different sea ice states, [Lohse et al., 2021] recommends the use of relatively large window sizes, as they lead to more accurate classifications overall. They suggest an optimal window size of w = 51 pixels for S1 EW imagery (i.e. 2040x2040m) and mention that window sizes of 21 and 11 pixels (i.e. 840x840m and 440x440m respectively) can be used as well, even though they lead to less accurate sea ice classification. Another study by [de Roda Husman et al., 2021] suggested the use of w = 11 for river ice classification on S1 IW imagery (a window size of 110x110m).

These studies were, however, not cloud-based, and therefore encountered different computational restrictions than in this study. It proved particularly tedious to sample large GLCM texture data sets for window sizes over w=21. As such our choice of w is a compromise between improving expected accuracy (large window size) and minimizing computational effort (small window size). Moreover, a downside of larger window sizes is the reduced spatial resolution that is introduced by the statistical computations over each GLCM window [Lohse et al., 2021]. As we aim to detect plume-driven polynyas that are mostly in the order of 1-10km2 [e.g. Gourmelen et al., 2017; Alley et al., 2019], we expect that a window size of more than 21x21 pixels (0.71 km2) would reduce the spatial resolution too much for it to be a useful window size for the purpose of this study.

In the end we computed GLCM textures for w = [5,11,21]. The addition of w = 5 is to assess how successful the choice of a relatively small, but computationally efficient window size to base a classifier on proves to be. As a GLCM feature is dependent on the window size w we compute these features for three different values of w and create different classifiers for each. In this way we eventually assess which choice of w is optimal for plume-driven polynya detection in chapter 4.

3.3. Classification

3.3.1. Classifiers

Multiple classifiers were developed. We start of with a base classifier, consisting of original HH and HV backscatter intensity as its input features. Three more classifiers were constructed, each using one of the three GLCM texture data sets as their input features. For clarity we will name the base classifier `BASE`. The classifiers for the GLCM data sets with window size w = [5,11,21] are named `GLCM5`, `GLCM11` and `GLCM21` respectively. We primarily focus on how well the classifiers perform at classifying the input images in water and ice. However, we also compare how they classify the images in their respective sub-classes to get an indication of how well they are able to distinguish rougher water from smoother ice.

3.3.2. Random Forest Classification

Several supervised classification algorithms are applicable for the detection of water bodies in sea ice, but research by Hoekstra et al. [2020]; Shelestov et al. [2017] has concluded that a Random Forest (RF) classifier is most useful for water-ice detection. Hoekstra et al. [2020] mentions that RF classification can be done with a multitude of input features, which is useful as we combine dual-pol SAR and GLCM features. Furthermore, an aspect of an RF classifier that is especially well-suited for this study is that the classifier is simultaneously able to assess feature importance [Hoekstra et al., 2020]. This allows for an in-depth analysis of optimal feature selection to improve classification accuracy. Shelestov et al. [2017] has analysed supervised machine learning algorithms in the GEE specifically and concluded that the GEE RF algorithm generally performs well and can be used as a reliable tool for classification.

In an RF classifier several hyper-parameters need to be specified. First of all, the classifier creates a prescribed number of decision trees, which are used predict a label for each pixel based on randomly sampled subsets of training data. For this, we manually define the desired number of classification trees (*numberOfTrees*). Furthermore, we define the minimal number of variables per split in a tree (*variablesPerSplit*), the minimum number of points in a training subset (*minLeafPopulation*) and the maximum number of nodes per tree (*maxNodes*). In Python, *GridSearchCV* from the *sklearn* package was used to test what configuration of these input parameters results in the highest accuracy score. This approach uses cross validation to determine the accuracy score of each combination of parameters and thus ensures that the validation data remains independent from the classification algorithm. When comparing 'optimized' hyperparameter choices from the *GridSearchCV* algorithm with 'standard' hyperparameter choices, we however concluded that the optimized hyperparameter choices resulted in similar or poorer classification results than the standard hyperparameter choices. We believe that the optimized values resulted in a classifier that was over-fitted on the training data and chose to use standard values as they resulted in a classifier that performed equally well, and was less prone to over-fitting. The following hyperparameter values were used: *numberOfTrees*=10, *variablesPerSplit*=2, *minLeafPopulation*=1, *maxNodes*=∞.

3.3.3. Accuracy Assessment

To assess how well each classifier performs, we compute each classifier's Cohen's kappa score [Cohen, 1960]. Cohen's kappa score is an accuracy metric that takes into account the full confusion matrix of a classifier. Where a normal accuracy metric is only dependent on true positives and true negatives, Cohen's kappa also depends on false positives (ice incorrectly classified as water) and false negatives (water incorrectly classified as ice), meaning that it gives an more complete view of a classifier's performance. Furthermore, the False Negative Ratio (FNR) and False Positive Ratio (FPR) of each classifier were computed. The FNR is the ratio of True Negatives to False Negatives, and is a measure to tell what fraction of water pixels are incorrectly classified as ice. Conversely, the FPR is the ratio of False Positives to True Positives and relates to the number of ice pixels incorrectly classified as water. As we wish to create a classifier that is specialized to detect isolated water bodies in frozen seas, we primarily aim to create a classifier with a FNR as low as possible.

To assess which classifier performed best, the kappa score of each classifier was compared to that of `BASE`. In this comparison, the comparative performance was expressed as an improvement score. This score is a fraction of the original kappa score, which was positive if the classifiers performs better and negative if the classifier produces worse results. The best classifier was chosen by looking at its improvement score, FNR, and by visually comparing classification results from each classifier in regions where polynyas, water leads, and smooth sections of sea ice are present.

Finally, we compared each classifier's confusion matrix for classification in main classes (*Water*, *Ice*) and for classification in sub-classes (*Open Water*, *Rough Water*, *Sea Ice* and *Floating Land Ice*) to be able to analyze which (sub-)class distinctions prove easy and difficult to accurately predict.

3.4. Post-Processing

The choice of smaller or larger window sizes will result in different strength and weaknesses of each classifier. Smaller window sizes will lead to a higher spatial resolution of the classification output [Lohse et al., 2021], but are expected to be less robust than one based on a larger window size. Plume-driven polynyas are characterised by an area of open water (1-10km2, [Alley et al., 2019]). Classified images of each classifier are expected to detect and classify pixels in these regions as relatively large and uniform bodies of water (>1km2). However, a fraction of incorrectly classified water pixels as sea ice is expected to occur (represented by each classifier's FPR). For these cases, we expect that the bodies of water detected are more irregular and, in general, small in size. We use this assumption to apply a filter based on the area of each detected body of water in the classified image. As small polynyas of approximately 1 km2 in size can occur, we apply a threshold of 1.2km2 (750 pixels of 40x40m) to filter out incorrectly classified water pixels.

To assess how post-processing affected classification, we analyzed the filtered images on a spatial basis. We compared each classifier result with and without applying an area filter and assessed which combination of classification and post-processing resulted in the optimal strategy for detecting plume-driven polynyas.

4

Results

4.1. Feature Selection

We applied RFECV to each texture set. An example for GLCM textures of *w*=11 is shown in Figure 4.1. The figure shows the accuracy of RF classification per *n* number of features. Plotted in red are the p-values that show how statistically dependent the classified labels are from the training labels. Using 16 features results in the best classifier for this data set. As the p-value for this choice of feature number is sufficiently high and shows no significant increase when using higher feature numbers, 6 features were chosen as a more practical choice. Feature importance scores of the top 6 features are given in Figure 4.1b and are in descending order of importance: Sum Average (HV polarization), Backscatter Intensity (HH polarization), Difference Variance (HV polarization), Dissimilarity (HV polarization), Difference Entropy (HV polarization) and Sum Average (HV polarization). We will from now on refer to these features as `HH_savg`, `HH`, `HV_dvar`, `HV_diss`, `HV_dent` and `HV_savg` respectively.

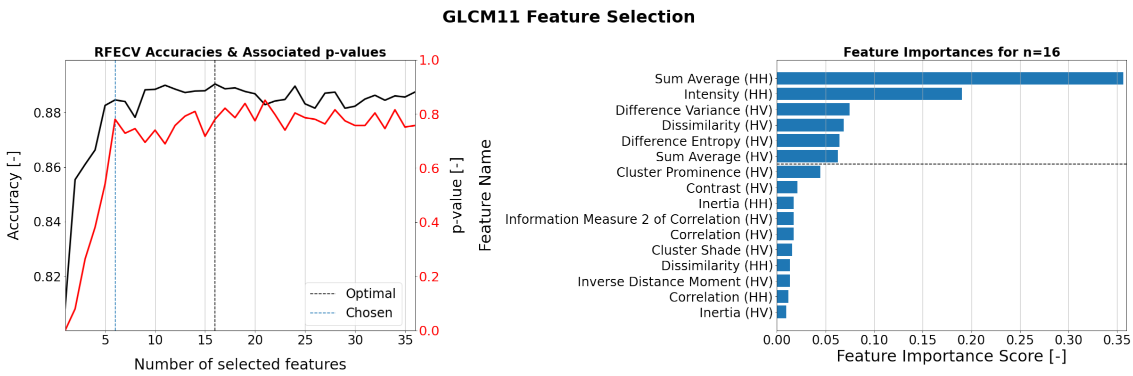


Figure 4.1: Feature Selection for `GLCM11`. Left: In blue, RFECV to select the optimal number of features and in red the student-t test to assess statistical independence for each number of features. Right: Feature importance for the chosen number of features (n=6).

Finally, the final choice of features was based on these features and their correlations. As `HV_dvar` was highly correlated to both `HV_diss` (96.0%) and `HV_dent` (90.1%) and as Figure 4.1b shows that `HV_diss` and `HV_dent` are of less importance than `HV_dvar`, `HV_diss` and `HV_dent` are both discarded. Similarly, `HH` was dropped as it correlates for 97.5% with `HH_savg` and the importance score of `HH_savg` is significantly higher. This left us with the final selection of features: `HH_savg`, `HV_savg` and `HV_dvar` to base `GLCM11` on. We repeated this process for the texture data sets of other window sizes and found that for both the data sets of `GLCM5` as well as `GLCM21`, `HV_dvar`, `HH_savg` and `HV_savg` were the optimal features to select. Results for these data sets are presented in Figure A.1 and A.2.

4.2. Feature Characteristics

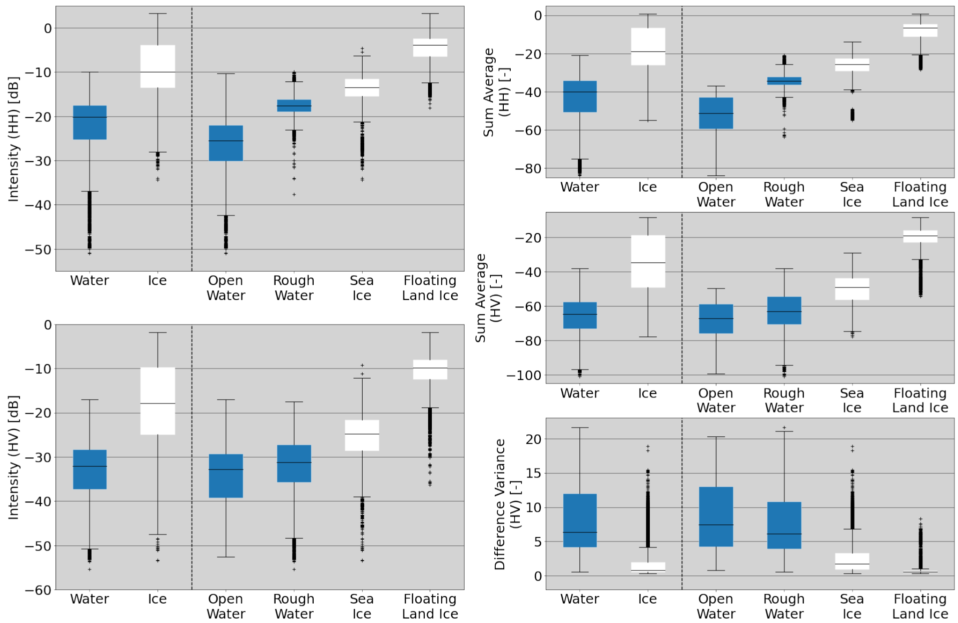


Figure 4.2: Boxplots showing the distribution of original backscatter intensities (left) and GLCM features (right) for GLCM11 training data

Figure 4.2 shows a spectral distribution of the training data per (sub-)class. The left boxplots show the input features of BASE, the original HH and HV backscatter intensities. The right boxplots show the input features of GLCM11. Boxplots of GLCM5 and GLCM21 are similar to results in Figure 4.2 and are presented in Figure A.6 and A.7.

In general, *Water* is characterized by low median values for HH (~-20 [dB]), HV (~-32 [dB]), HH_savg (~-40 [-]) and HV_savg (~-65 [-]). For calmer open water low intensities in HH and HV polarization are expected due to surface scattering and low surface roughness. On the other hand, *Water* shows high median HV_dvar values (~6 [-]), even though the spread in values is large (~0.1 to 22 [-]). Conversely, spectral characteristics for ice are in a general sense opposite to those of water for each feature. Rough(er) surfaces of types of *Ice* and more volumetric scattering compared to *Water*, result in higher median values for HH (~-10 [dB]), HV (~-18 [dB]), HH_savg (~-20 [-]) and HV_savg (~-35 [-]). *Ice* has a very low median value for HV_dvar, at ~0.1 [-]. HV backscatter intensity is extremely low for *Water* and its sub-classes *Open Water* and *Rough Water*. Their range goes down to less than -50 [dB], which is well below the noise floor of the cross-polarization sensor (-40 [dB], ESA [2012]). Values lower than the noise floor are considered as outliers, but are kept in the training data as we expect RF classification to handle the presence of outliers well.

When looking at the spectral distribution of *Water* and *Ice*, the difference between the two classes can appear to be quite distinct for both HH and HV as median values are significantly different. However, Figure 4.2 also shows that this is due to the influence of the more easily distinguishable sub-classes of water and ice: *Open Water* (median HH value ~-25 [dB], median HV value ~-33 [dB]) and *Floating Land Ice* (median HH value ~-5 [dB], median HH value ~-10 [dB]). *Open Water* is characterized by low values of HH_savg and HV_savg, due to its smooth specular surface. *Floating Land Ice*, shows the opposite behaviour. Values in HH_savg and HV_savg are higher than those of Water due to its rough surface and its composition of mostly snowy surfaces, leading to higher backscatter intensities. The sub-classes *Rough Water* and *Sea Ice* are a lot more ambiguous regarding their median intensities in both HH (~-18 [dB] and ~-14 [dB] respectively) and HV (~-31 [dB] and ~-26 [dB] respectively). This ambiguity is still present in HH_savg (~-35 [-] and ~-25 [-] respectively) and HV_savg (~-65 [-] and ~-50 [-] respectively). The HH_savg and HV_savg values of *Sea Ice* have similar standard deviations as those of *Rough Ice*, but its median values are higher. The spread in intensities is considerable as HH_savg and HV_savg intensities are dependent on the particular water and sea ice states at the moment of data acquisition. Some sea ice states are smoother, less snowy, or wetter than others. These sea ice states reflect few radar waves back to the sensor, since more surface scattering takes place, and surface roughness is more comparable to that of *Rough Water*.

HV_dvar shows its strength in its ability to separate the *Rough Water* from the *Sea Ice*, as median values for both sub-classes differ significantly (~6 [-] and ~1 [-] respectively). This is reflected in Figure 4.2, as HV_dvar achieved third highest importance score behind HH_savg and HH. Opposed to HH_savg and HV_savg the median value of HV_dvar for *Sea Ice* is relatively low. The spread in values however is much higher than that of Floating Land Ice, as state of sea ice for *Sea Ice* can become near as smooth as some (rougher) water surfaces. Still, all texture features show overlap for each sub-class (*Rough Water* and *Sea Ice* in particular) outside of the 1st and 3rd quantile, meaning that fully accurate classification remains a challenge and classification errors are still likely to occur.

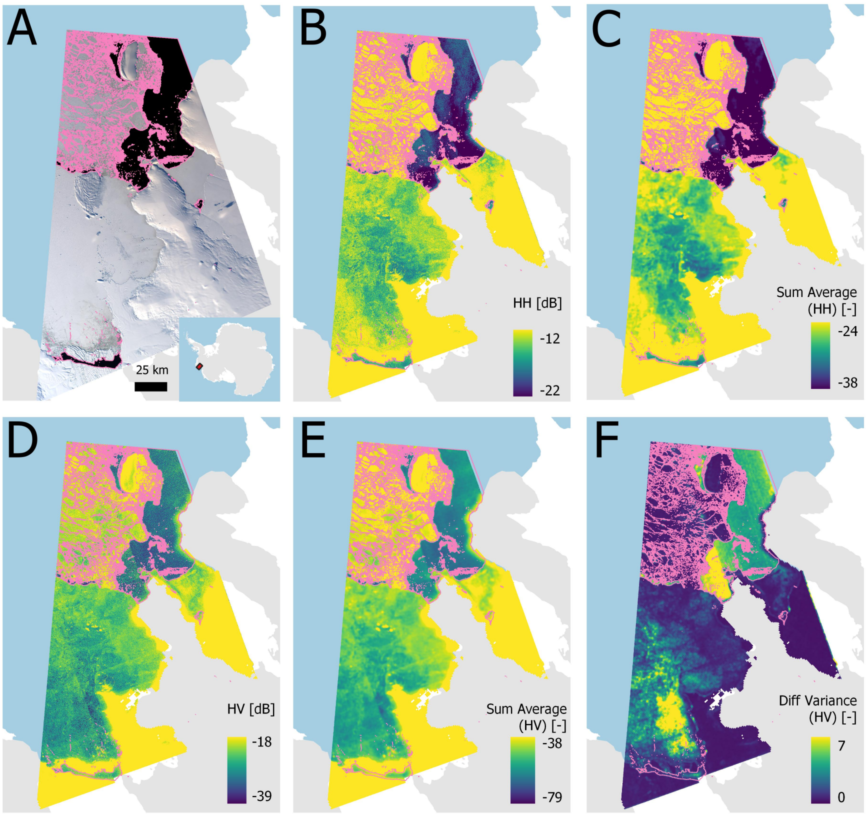


Figure 4.3: HH, HV and GLCM21 texture features for *V2*. Optical image is added for visual reference. For clarity, we highlighted water extent in the optical image by overlaying its water-ice border on top of every image in pink.

We visualized each GLCM21 texture feature for *V2* in Figure 4.3 for visual interpretation. For reference, texture features for *V1*, *V3*, *V4* are presented in Figure A.3, A.4, A.5. We show GLCM21 features as these show the largest contrast with the original HH and HV features, making them best-suited for visual comparisons. First off, when comparing HH (4.3B) and HV (4.3D) to HH_savg (4.3C) and HV_savg (4.3E), we notice the smoothing effect of the statistical matrix computations on original input features. In large sea ice or open water regions, the feature data of HH_savg and HV_savg are much more uniform and show significantly less variation in magnitude within such regions. In Figure 4.3E we see that large open water bodies are associated with higher values of HV_dvar (in accordance with Figure 4.2, although we see similar pattern in specific regions of sea ice. It appears that smaller bodies of water are not necessarily associated with higher HV_dvar values. To highlight this, we zoom in at three specific regions of small scaled water bodies in Figure 4.4. Smaller bodies of water do not consistently show high values of HV_dvar as the water in bodies in 4.4V2-1 and 4.4V2-3 both show lower ranged HV_dvar values, whilst 4.4V2-2 is associated with higher values. For bodies of water, the spread in HV_dvar is therefore significant (see also Figure 4.2). Furthermore, we see an edge smoothing effect of GLCM21 features in 4.4V2-1 and 4.4V2-3. Where *FLoating Land Ice* is present next to *Water*, high values of HH_savg and HV_savg are found in the original open water body with a width comparable to window size *w*=21. In 4.4V2-2 an edge pattern is also present. 4.3V2-2D (and 4.3V2-2E, but slightly less apparent) shows that a low intensity edge extends outwards from the water body on the western side, while a high intensity edge protrudes inwards into water body's eastern edge. GLCM5 and GLCM11 data show similar results, but the spatial extent of edge smoothing is less significant.

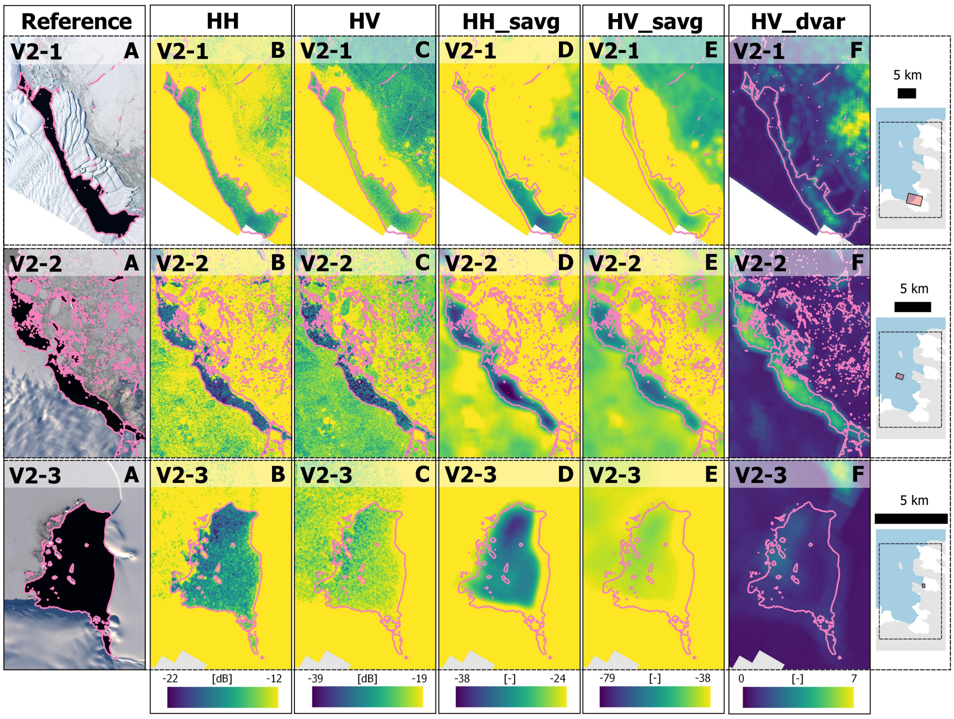


Figure 4.4: HH, HV and GLCM21 texture features zoomed in at small scale water bodies for *V2*. A water mask from the optical reference image is overlayed in pink for convenience.

4.3. Classifier Performance

4.3.1. Accuracy Assessment

Table 4.1 shows how each classifier performed. To assess each classifiers ability to classify polynyas correctly as water, we denote FNR, FPR and the kappa score apart from overall accuracy. A high FNR indicates a poor ability to correctly classify water, whereas a high FPR indicates an inability to detect ice. A low FNR is thus essential to detect plume polynyas, as they are isolated bodies of water surrounded by ice. The kappa score takes both ratios into account for its accuracy, but we also evaluate whether a low kappa score corresponds to the lowest FNR.

Table 4.1: Classifier performance scores and comparative performance to BASE for water-ice classification.

Classifier	Spatial Resolution [m]	Results				Improvement from BASE [%]
		Accuracy [-]	Kappa [-]	FNR [%]	FPR [%]	
BASE	40	0.922	0.844	9.39	6.22	-
GLCM5	200	0.959	0.915	4.74	3.72	+8.46
GLCM11	440	0.963	0.927	4.49	2.83	+9.81
GLCM21	840	0.965	0.930	4.25	2.78	+10.2

Each classifier has a water-ice classification accuracy above 92%, but the kappa score of `BASE` of 84% is significantly lower than any of the kappa scores of the GLCM classifiers, which are all higher than 91%. We can see that the accuracy of `BASE` gives a false sense of high quality classification as both FNR and FPR are around 2 times higher than those of the GLCM classifiers. Therefore, each GLCM classifier is a significant improvement compared to `BASE`, also denoted by their respective improvement scores. `GLCM21` performs best, with a kappa score of 93.0% (+10.2% improvement from `BASE`). `GLCM11` has the second-highest kappa score, at 92.7% (+9,81% improvement from `BASE`). `GLCM5` scores lowest, with a kappa score of 91.5% (+8.46% improvement from `BASE`). In terms of FNR and FPR, we see a similar pattern. Of the GLCM classifiers, `GLCM21` has lowest FNR and FPR scores at 4.25% and 2.78% respectively, then `GLCM11` with 4.49% and 2.83% and lastly `GLCM5`, which has a FNR of 4.74% and a FPR of 3.72%. Kappa scores, FNRs and FPRs of `GLCM11` and `GLCM21` are almost equal, whereas those of `GLCM5` deviate relative to these classifiers.

4.3.2. Confusion Matrices

Confusion matrices provide another means to visualize classifier accuracy and portray how accurately each (sub-)class is predicted. In Figure 4.5 we present the confusion matrices of each classifier per class and in Figure 4.6 confusion matrices for sub-classes are given.

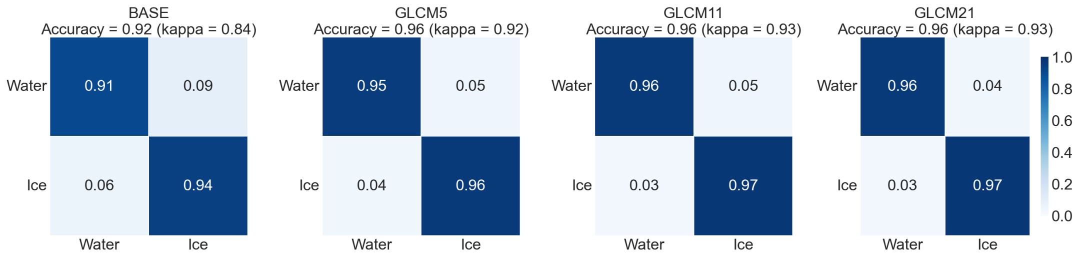


Figure 4.5: Confusion matrices of each classifier for water-ice classification.

Figure 4.5 shows the same performance pattern as the values of FNR and FPR in Table 4.1. *Water* and *Ice* pixels are classified correctly for >91% for each classifier, with percentages reaching 96% and 97% respectively for `GLCM11` and `GLCM21`. Comparing the GLCM classifiers, we can again see that `GLCM5` performs slightly worse than both `GLCM11` and `GLCM21`, as true positives and true negatives are marginally lower, whilst false negatives and false positives are marginally higher.

In Figure 4.6 we take a closer look at each classifier to see which sub-classes prove most difficult to accurately detect. For sub-class classification, the accuracy pattern found in Table 4.1 and Figure 4.5 is repeated again. `BASE` performs significantly worse than the GLCM classifiers (accuracy = 0.81, kappa = 0.74), with `GLCM21` being the most accurate classifier (accuracy = 0.87, kappa = 0.83). Performance of `GLCM21` equalled by `GLCM11` (accuracy = 0.87, kappa = 0.83). Somewhat less accurate is `GLCM5` with an accuracy of 0.86 and kappa of 0.81. When looking at each confusion matrix it is clear that each classifier has most difficulty with accurately distinguishing the *Water* sub-classes. Accurate detection of *Ice* sub-classes proves more effective. Finally, as expected, each classifier experiences relatively high difficulty with accurate distinction of *Rough Water* and *Sea Ice*. Every GLCM classifier shows the same performance in incorrectly classifying *Rough Water* as *Sea Ice* (False Negatives), each having omission errors of 10%. Although substantial in magnitude, omission errors for these sub-classes are significantly reduced compared to `BASE` (omission error = 15%). For the incorrect classification of *Sea Ice* as *Rough Water* (False Positives), `GLCM5` is marginally less accurate (omission error = 8%) compared to both `GLCM11` and `GLCM21` (omission error = 5%).

4.4. Window Size Analysis

In Figure 4.7 we compare the effect of window size choices to the classification outputs on all four validation images. Figure 4.7, 4.8a and 4.8b show that every classifier, `BASE` as well, is able to distinguish very large bodies of water from ice regions. However, `BASE` classification also leads to very high number of False Positives and False Negatives within classified regions of *Water* and *Ice*. Large portions of sea ice show regions of classified water bodies, seen in Figure 4.7V1-4. Similarly, the figure shows

Figure 4.6: Confusion matrices of each classifier for sub-class classification.

a multitude of smaller classified ice bodies within the larger open water regions. Compared to `BASE` output, all the GLCM classifiers perform significantly better. Each classifier is able to pick up bodies of water whilst keeping the FPR and FNR substantially lower.

For an in-depth comparison of each GLCM classifier, we zoom in to examples of open water (Figure 4.8a) and sea ice (4.8b) respectively to better illustrate the performance of the different GLCM classifiers. When looking at Figure 4.8a, we notice that `GLCM21` performs relatively poorly in classifying small bodies of water, as the classifier misses water near edges of open water bodies. `GLCM5` and `GLCM11` perform better at detecting the edge of water bodies and can detect smaller bodies of water. In Figure 4.8a (row A), near edges and in the smallest water leads, we primarily see yellow masks, meaning that these regions of water are only picked up by `GLCM5` (most clearly visualized in Figure 4.8aV4A). Slightly further away from the sea ice edges, we primarily see green masks, meaning that at this distance `GLCM11` also detects water (e.g. Figure 4.8aV2-2A, 4.8aV2-3A and 4.8bV1-2A). Only when we are significantly farther away from a sea ice edge, the mask is purple, meaning the `GLCM21` also detects water (Figure 4.8aA). From visual interpretation, `GLCM5` therefore clearly has the best performance for this specific aspect of classification, which should be reflected in a low FNR. However, the FNR does still decrease from `GLCM5` (FNR = 4.74%) to `GLCM21` (FNR = 4.25%), even though the difference is relatively minor (-0.25% and -0.49% respectively). The FNR therefore cannot be used on its own as a reliable assessment of a classifier's ability to detect water.

Conversely, `GLCM5` performs poorest at classifying sea ice correctly, resulting in a high number of pixels falsely detected as open water (higher FPR). The FPR increases significantly for `GLCM5` (FPR = 3.72%) compared to both `GLCM11` (FPR = 2.83%) and `GLCM21` (FPR = 2.78%). Figure 4.8b shows that `GLCM21` indeed leads to the least amount False Positives pixels. In Figure 4.8bV2 and 4.8bV4 differences in sea ice detection are clear. Comparing each classifier in 4.8bV2-A and 4.8bV4-A shows that `GLCM5` has a much higher FPR for sea ice regions than `GLCM21` (large portions of sea ice are

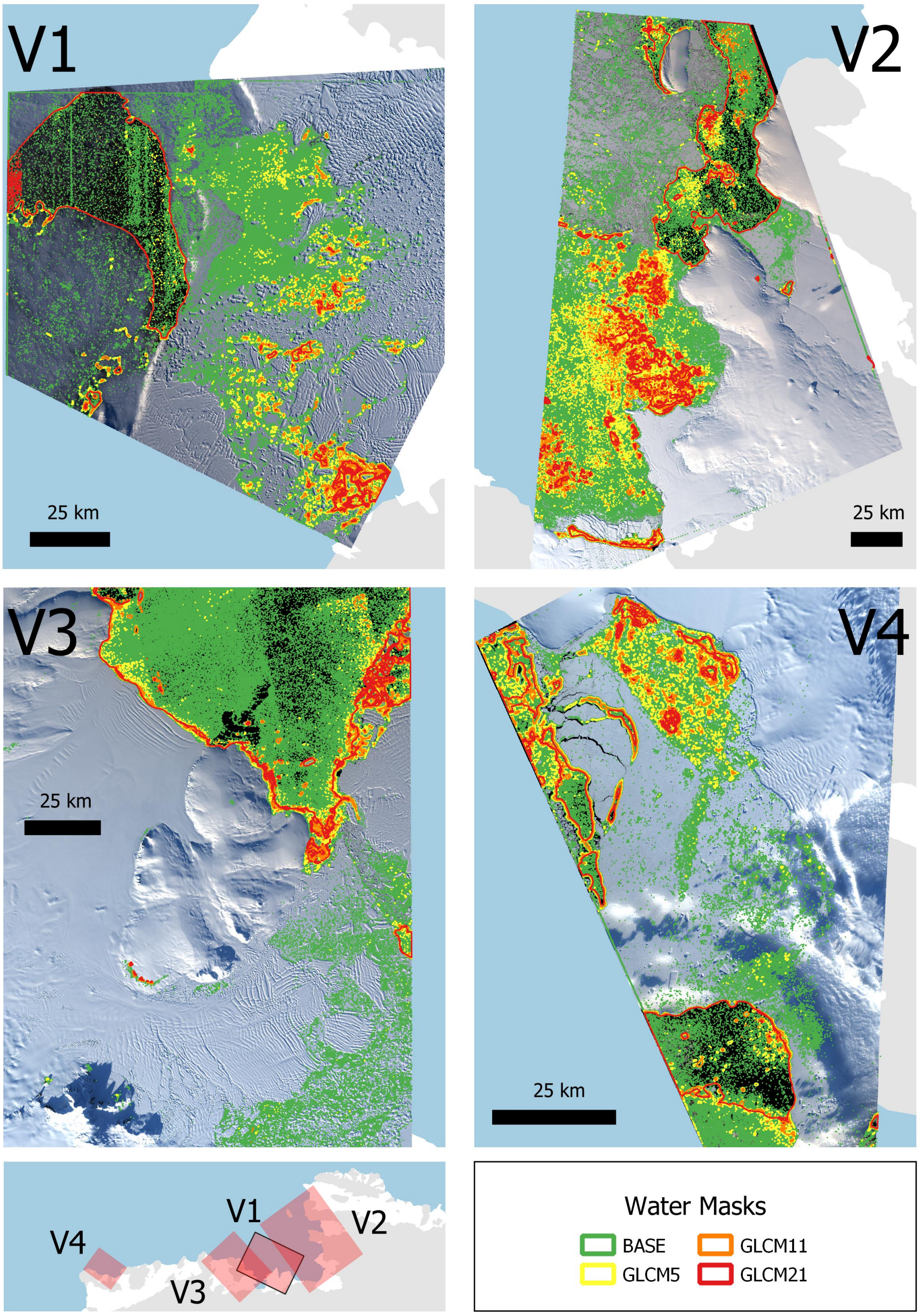
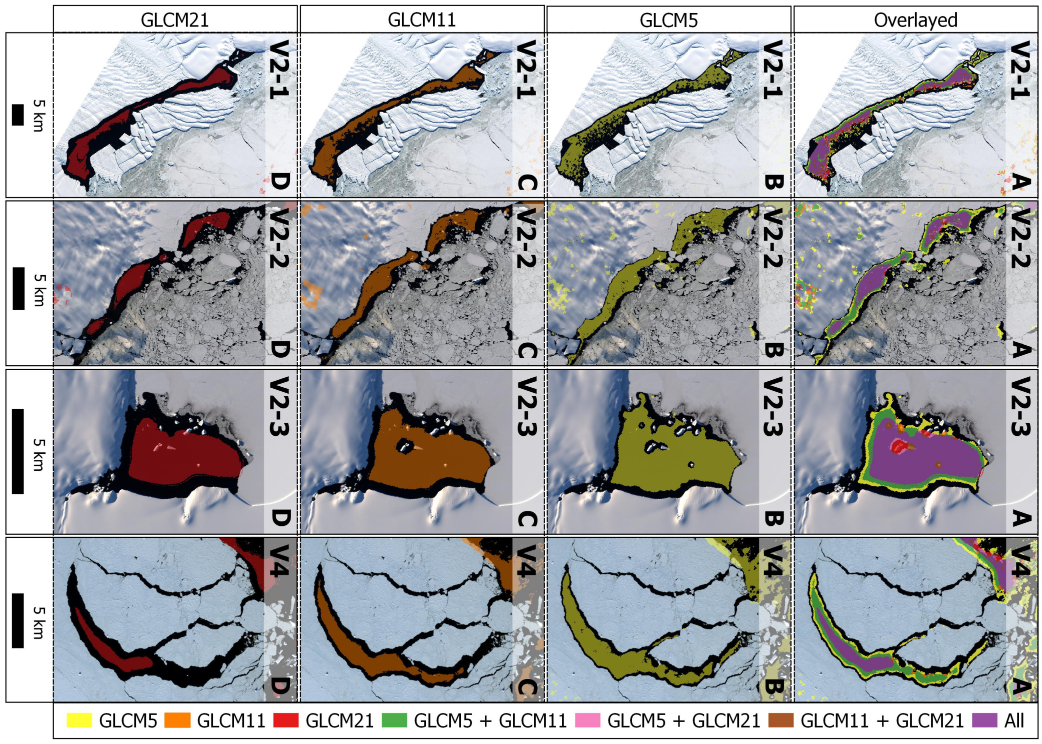
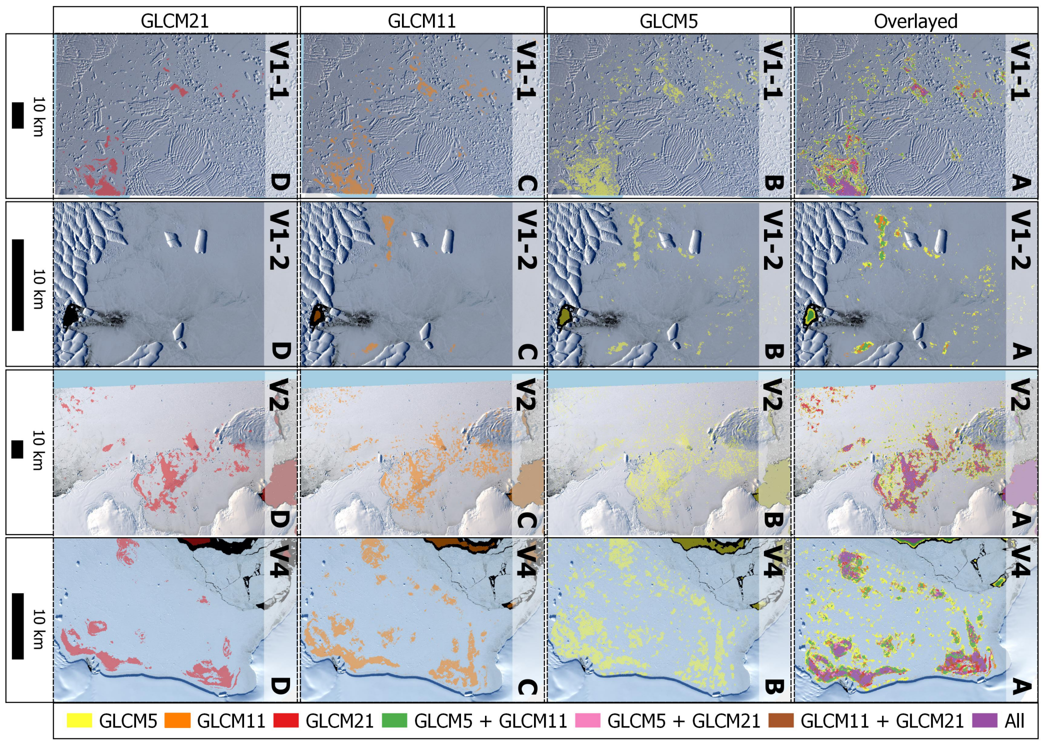


Figure 4.7: Window size influence on water-ice classification. On top of an optical reference image, classification outputs of each classifier are presented as water outlines. `BASE`: Green, `GLCM5`: Yellow, `GLCM11`: Orange, `GLCM21`: Red.

overlayed by the yellow mask). `GLCM11` shows substantially less False Positives than `GLCM5` (orange and green masks), but shows some extra False Positive regions compared to `GLCM21` (red and purple masks). In terms of sea ice classification, FNR values presented in Table 4.1 are in accordance with spatial patterns found in Figure 4.8b.



(a) Polynyas and water leads



(b) Sea Ice

Figure 4.8: Classification output for different window sizes, zoomed in at regions of interest.

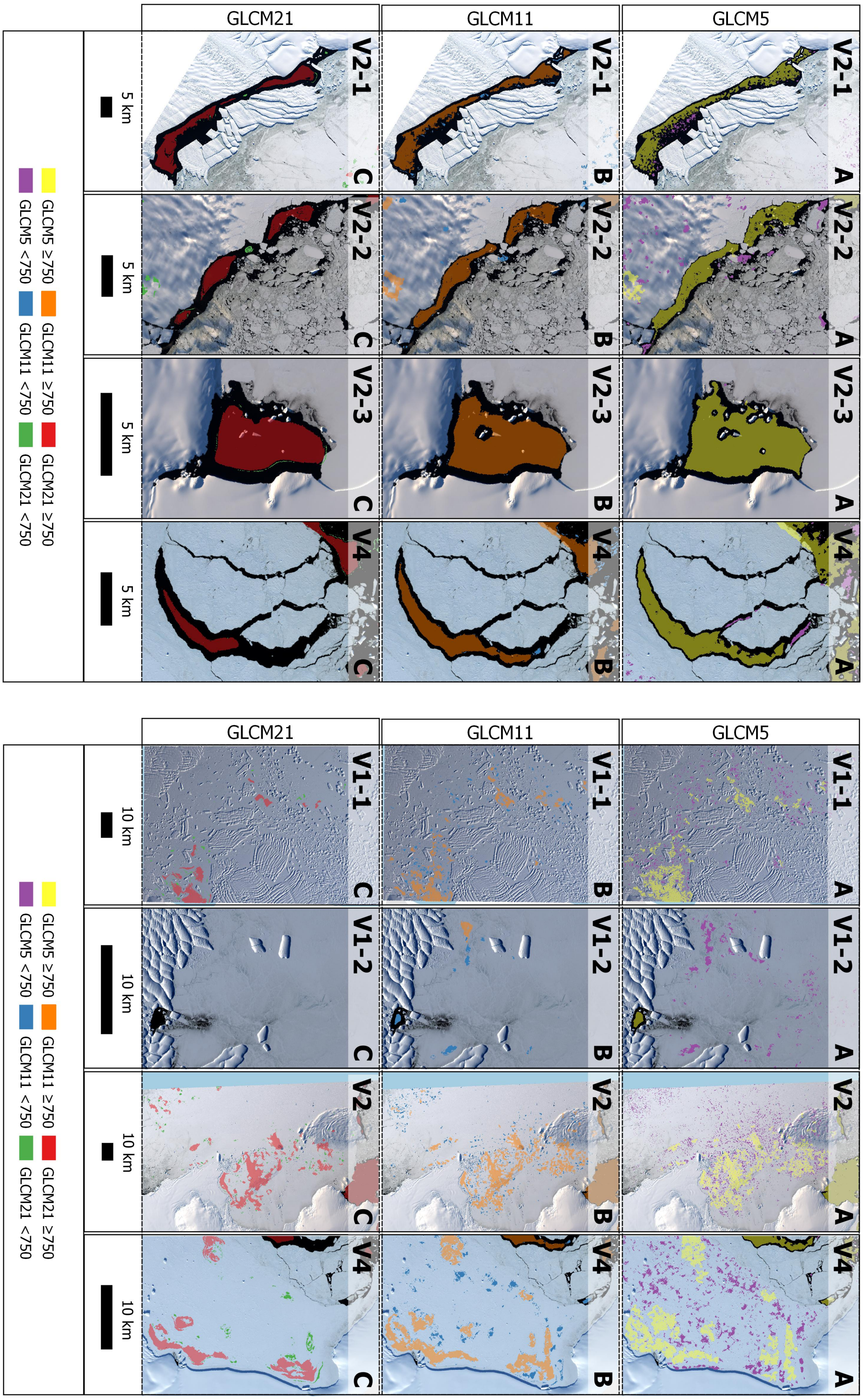
4.5. Post-Processing

Figure 4.5, Figure 4.8a and Figure 4.8b have shown that increasing GLCM window size leads to a significantly lower FPR, but leads to poorer performance in accurate small water body and water edge detection. Ideally, we merge the ability of `GLCM21` in minimizing FPR with the ability of `GLCM5` to detect water bodies up to small spatial scales. We try to achieve this by applying an area filter on classified bodies of water.

To analyse the influence of filtering the classified open water bodies by a minimum area of 1.2km2 (750 pixels of 40x40m), we look at classified water bodies in regions of sea ice and regions where polynyas and smaller water leads occur. We visualize water bodies that are included and excluded by this filter for each classifier in Figure 4.9a and 4.9b. For a complete overview of the filtering step on each validation image we have visualized this filter for each classifier in Figure A.8a, A.8b and A.8c.

Looking at Figure 4.9b, we see very little effect of filtering on `GLCM21` classification outputs. For `GLCM21` not many incorrectly classified bodies of water are smaller than 1.2km2, meaning that few of these water bodies are filtered out. Only in Figure 4.9bV4-C a significant positive effect is visible on removing False Negative pixels. The effect for `GLCM11` is more pronounced. We see a positive effect in Figure 4.9bV4-B, where a large fraction of False Positives are filtered out. However, the negative effect of this filter step is visible in Figure 4.9bV1-2B. When applying this filtering step, the output of `GLCM11` does not contain a small scaled polynya anymore. When filtering the output of `GLCM5`, we again see that a significant amount of False Positive pixels are filtered from the results in Figure 4.9bV4-A. But, even after filtering, the amount of False Positives is significantly higher than `GLCM11` and `GLCM21` especially, although the False Positives are now mostly found in the same regions of sea ice, where they were first much more spread out. Finally, the filtering step does not filter out the small polynya from the `GLCM5` result, seen in Figure 4.9bV1-2A, meaning that filtering `GLCM5` classification outputs does not impact its ability to detect water bodies below 1.2km2 as significantly as for `GLCM11` and `GLCM21`.

When comparing effects of filtering on classification outputs near sea ice regions in Figure 4.9a, we see similar results. In 4.9aV2-3C and 4.9aV4C no bodies of water are filtered from the `GLCM21` ouputs, whereas in 4.9aV2-1C and 4.9aV2-2C the number of removed bodies of water is minimal (roughly 3-5). We do however see that those bodies of water that are removed, correspond to relatively large water leads or polynyas. The fact that only a small portion of such water bodies is detected by `GLCM21` is due to the reduced spatial resolution by choosing a larger window size, as described before. We see that `GLCM21` already has difficulty in detecting smaller scale water features (especially in 4.9aV4C), and applying this filter step slightly increases the already relatively poor performance of the classifier. The effect of applying the filter step on `GLCM11` images is slightly more pronounced, but still marginal. A couple of water bodies in 4.9aV2-1B, 4.9aV2-2B and 4.9aV4B are removed, but they do not influence the extent of classified water bodies significantly. As `GLCM11` has a higher spatial resolution, it is able to better detect water edges and smaller wakes than `GLCM21`. As such, single water bodies are not detected as multiple bodies of water (4.9aV2-2C compared to 4.9aV2-2B) and filtering does not take place. Finally, `GLCM5` shows the most filtering of small water bodies, as it is able to detect the smallest scale water bodies. Especially in 4.9aV4A we see that two smaller water leads that are correctly classified as water are removed with this filtering step. Still, despite filtering such bodies of water, `GLCM5` provides more accurate water body detection than `GLCM11` and in particular compared to `GLCM21`.



(a) Filtering Polynyas and water leads

(b) Filtering Sea Ice

Figure 4.9: Overview of impacts of filtering smaller classified bodies of water (750 pixels) from the classification output

5
Discussion

5.1. Classification

In both Table 4.1 and Figure 4.7 we compared the ability of each classifier to detect open water bodies. We could see that BASE performs significantly poorer than any GLCM classifier, as its kappa score is significantly lower, and both its FNR and FPR are higher by roughly factor two to three. Figure 4.7 has shown the effect of this lower accuracy in a spatial context. The water classification of BASE is much less uniform compared to GLCM5, GLCM11 and GLCM21. Large numbers of False Negative pixels are visible even in greater bodies of water, indicating that BASE is significantly less robust in its ability to distinguish open water regions from sea ice. False Positive pixels are much more prevalent across sea ice regions in all validation images, showing a similar reduced robustness for accurate distinction of sea ice compared to rough water. Overall, BASE results in a noisy water mask, while sea ice is detected much too inaccurately to be able to use this classifier effectively. Our results confirm that HH and HV alone are ill-suited for accurate water-ice classification and plume-driven polynya detection.

Figure 4.2 (plus Figure A.6 and Figure A.7) have shown us that, after filtering out features that are correlated for more than 90% to each other, HH_savg, HV_savg and HV_dvar are the unique features that have the largest potential for accurate water body detection based on their importance scores. Figure 4.2 and Figure 4.3 respectively show the added value of each feature from a data perspective and from a spatial context. In Figure 4.2 we saw that the importance of HH_savg and HV_savg is always higher than that of HH and HV. These importance scores are likely attributed to the fact that textural features pick up spatial features in the GLCM window: HH_savg and HV_savg (and HV_dvar) make use of extra information compared to the original backscatter intensity features HH and HV. This, however, is at the expense of spatial resolution. Optionally, instead of three, the six 'best' GLCM texture features (see Figure 4.1) could have been used to, in this case, base GLCM11 on. We argue that this would have likely only improved classification accuracy marginally, as removed texture features do not provide a significant amount of extra information due to their high correlation with HH_savg or HV_dvar. Despite high correlations, we do however argue that the classifier would likely have been fractionally more robust, as extra information of texture features with high importance improves the training process. However, by only selecting HH_savg, HV_savg and HV_dvar, we were able to do a more in-depth analysis of the influence that each texture feature has on classification, through which we were able to understand the classification process more thoroughly.

In Figure 4.3 we saw the effect of our choice for using the GLCM features rather than original backscatter intensity values. In regions of open water this choice aids accurate classification, as values of HH_savg and HV_savg are more uniformly low. In sea ice regions this smoothing effect can have an adverse effect if too many pixels occur within the GLCM window with a significantly low intensity. The smoothing effect in HH_savg and HV_savg can then lead to patches of sea ice with consistently and uniformly low intensity values, which can explain the larger regions of False Positives. Even though False Positives are introduced in this manner, the overall effect of using GLCM features is positive: it leads to significantly lower FNR and FPR for each GLCM classifier with respect to BASE (Table 4.1). A downside of using GLCM features is the smoothing or translation of sharp contrasts in HH and HV values at the water-ice shoreline (see Figure 4.4). A sharp transition from high to low (or low to high) HH and HV intensities within the GLCM window, leads to a displacement of this transition in GLCM features of, at most, the width of the GLCM window. This impacts classification accuracy negatively, as original backscatter intensity transitions will always be adjusted. In some cases, the water-ice boundary will be extended outwards, increasing the classified water body area, while in others, the boundary was translated into water bodies, thereby decreasing its classified area. When *Floating Land Ice* borders *Water*, as is often the case for plume-driven polynyas, very high GLCM feature intensities (HH_savg and HV_savg) for *Floating Land Ice* are directly adjacent to low *Water* feature intensities. During GLCM computation, this results in a translation of the original water-ice boundary already when only a few *Floating Land Ice* pixels are present within the GLCM window (as seen by an extension of high intensity values roughly 21 pixels (2040m) wide in 4.4D&E) within the body of water. Consequently, using GLCM features introduces a bias towards the detection of sea ice in such regions.

As mentioned before, classification results are heavily dependent on training data. Sampling is therefore a crucial aspect of this research. Given the nature of this research, sampling was a random selection process of S1 input data, with labels being assigned manually. The training process could have been improved by having in-situ measurements of the surface for each (sub-)class. As this was not the case, we believe that, in this way, errors were introduced in the classification output.

As the added value of creating a GLCM classifier using HH_savg, HV_savg and HV_dvar as input features has been made apparent, the most important variable to assess next is the GLCM window size *w*. Table 4.1 shows a correlation between combined kappa, FNR and FPR scores and window size. By decreasing spatial resolution and increasing the GLCM computation window kappa increases consistently (GLCM5: 91.5%, GLCM11: 92.7%, GLCM21: 93.0%), while FNR (GLCM5: 4.74%, GLCM11: 4.49%, GLCM21: 4.25%) and FPR (GLCM5: 3.72%, GLCM11: 2.83%, GLCM21: 2.78%) decrease consistently. Using a window size of *w*=5 results in an improvement score of 8.46%. Roughly doubling the window size *w*=11 increases the improvement score to 9.81% (+1.35%), which is considerable. Quadrupling the window size to *w*=21 leads to a marginal extra increase of the improvement score (+10.2%, which is +1.74% compared to GLCM5 and +0.39% compared to GLCM11). Increasing *w* is therefore beneficial for accurate classification, but as the increase in accuracy between GLCM11 and GLCM21 is marginal, we argue that increasing *w*>11 is not worth the extra decrease in spatial resolution (440m to 840m). To confirm this, we refer to the spatial context of the classification results in Figure 4.8a and 4.8b.

From spatial interpretation, increasing window size shows a clear increased performance in the detection of open water. GLCM5 performs best at detection small scale water bodies, GLCM11 and GLCM21 both perform worse. GLCM21 has most difficulty with detecting the full extent of small scaled water bodies. Significant water sections are not detected by the classifier, most prevalent for water leads with sub-kilometer widths (Figure 4.8aD). This effect is repeated for GLCM11 (4.8aC) and GLCM5 (4.8aB), but at a much lesser extent than for GLCM21. While GLCM5 clearly outperforms GLCM21 in terms of minimizing False Negatives for small water bodies, this is not reflected in Table 4.1 (GLCM5: FNR = 3.72%, GLCM21: FNR = 2.78%). Furthermore, looking at Figure 4.7, we see no significant difference between the two classifiers in False Negatives within large bodies of water (as is the case for BASE). This shows that the overall FNR computation in Table 4.1 is somewhat biased towards large open water bodies and does not accurately represent the behaviour we see in Figure 4.8a. This bias is most likely introduced due to the random selection of sample points. Perhaps by chance, significantly less water pixels are sampled near the ice-water edge and within sub-kilometer scaled water leads, meaning that incorrect detection is not picked up.

With respect to sea ice detection, we see the opposite correlation between *w* and detection performance. By increasing window size, GLCM11 and GLCM21 are much better at classifying large sections of sea ice correctly than GLCM5, as the standard deviation of HH_savg, HV_savg and HV_dvar intensities are significantly lower than those of HH and HV for both water and ice regions. This is reflected in the fact that considerable portions of Figure 4.8bA are masked in yellow (GLCM5 outputs). Most of these classified bodies of water are small in size. They are detected as water as the window size is not large enough to smooth out high intensity values from (predominantly) low intensity values. Similar patterns are visible for GLCM11 by looking at green and orange masks (4.8bA). Note that for the output of this classifier, areas of False Positives are larger than those of GLCM5, due to a larger window size. Exactly due to this effect, GLCM21 detects the least amount of water bodies on sea ice, but the water bodies that are detected are considerably large in size (4.8bA, red and purple masks). We thus note that, although significantly better than BASE, every GLCM classifier shows an inability to classify several (considerably large) sections of sea ice in Figure 4.8bV1-1, 4.8bV2 and Figure 4.8bV4 correctly.

We thus argue that this inherent disadvantage to using dual-pol SAR data for water-ice classification cannot be overcome by using GLCM texture based classifiers alone.

Overall, a pattern emerges. Increasing window size leads to lower overall FNR and FPR, at the cost of spatial resolution. What spatial analysis clearly shows though, is that this reduction in spatial resolution has significant impacts on what type of water body a classifier is able to detect. Applying a window size *w* for GLCM computation leads to an inability to detect water edges and bodies of water at spatial scales that are smaller than this reduced spatial resolution. `GLCM21` performs best in identifying sea ice, showing minimal False Negatives. `GLCM11` performs comparable to `GLCM21` on this front. However, its spatial resolution is twice as high, meaning that simultaneously, it is better able to detect small scale water bodies. Thus, `GLCM11` outperforms `GLCM21` in open water detection. `GLCM5` results in the most accurate open water detection. With a spatial resolution of only 200m this classifier can detect significantly smaller water leads and can detect the water-ice edge of polynyas much more accurately than both `GLCM11` and `GLCM21` in particular. `GLCM5` performs however extremely poorly in sea ice detection, introducing significant numbers of False Positives in the classification output. Its FPR is significantly higher than that of `GLCM11` and `GLCM21` and is so high that despite its accurate open water detection, it is overall ill-suited for plume-driven polynya detection in its current form. The same holds for `GLCM21` due to its low spatial resolution. As such, we argue that `GLCM11` is best suited for accurate plume-driven polynya detection, based purely on classifier performance.

5.2. Post-Processing

In chapter 4 we show effects of filtering each classifier's output on the minimum water body size of 1.2km2 (750 pixels of 40x40m) and assess whether False Positives can be substantially and accurately reduced for any of the classifiers, while minimizing the amount of True Positives that are removed. Filtering particularly affects the `GLCM21` output in a negative sense, as smallest regions of correctly detected water (True Positives) in its output are removed. For `GLCM21` this removal is a substantial impact as `GLCM21` already has difficulty with detecting smaller water bodies. For `GLCM5`, sections of classified water are filtered out as well. However, these detected water bodies are of a much smaller scale than those of `GLCM21` and we therefore argue that these True Positive removals are acceptable losses. From the output of `GLCM11` fewer True Positives are removed compared to `GLCM5` and `GLCM21`, as almost all of the classified water bodies in each region are larger than 1.2km2. The exception to this is the small (possibly plume-driven) polynya in 4.9bV1-2. When filtering the `GLCM11` output, detection of this polynya is removed, while in filtered `GLCM5` results this polynya is preserved.

Over sea ice regions, filtering small scale water bodies shows a considerable positive impact. Here, False Positives in classification outputs of every classifier are present and are removed systematically by the filter. The effect is most prevalent when filtering `GLCM5` output, as lower window sizes result in higher spatial resolution of the classification result, leading to many False Positive regions which are smaller than 1.2km2 (4.9bV4-A). A significant amount of False Positive regions are removed in `GLCM11` as well, although the increased window size of classification results in a lower impact of this filter on the amount of False Positives removed (4.9bV4-B). Some False Positives are removed from `GLCM21` results, but the effect is far less significant than for `GLCM5` in particular, as the use of a larger window size already smoothed out most small scale features (<1.2km2) during classification. In every filtered classification result, multiple regions of False Positives of considerable size still remain, which, irrespective of window size, cannot be removed with an area filter.

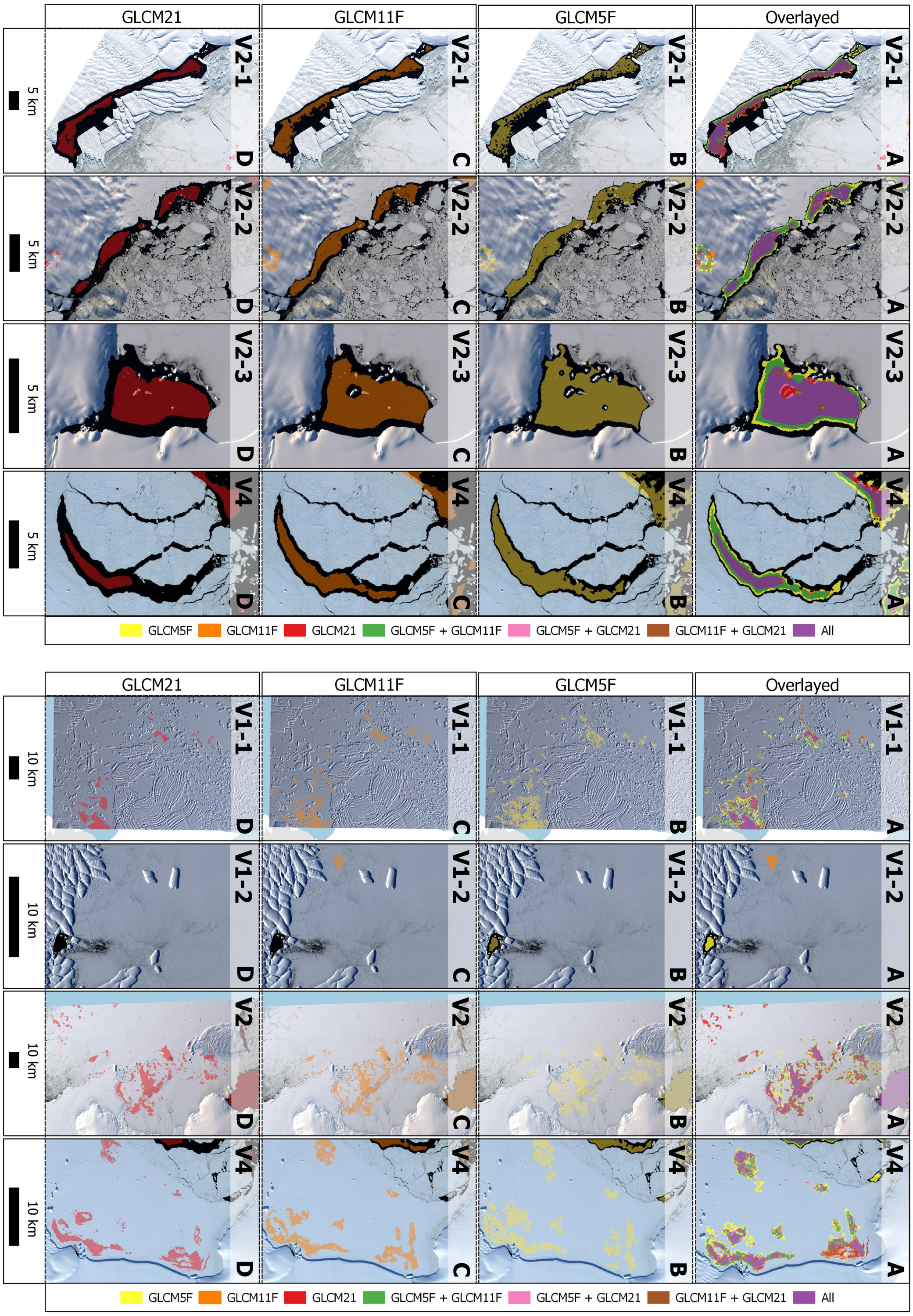
Overall, we suggest to avoid filtering `GLCM21` results to prevent True Positive removal from classification output that already suffers from poorer water detection. We suggest to filter both `GLCM11` and `GLCM5` outputs for areas over 1.2km2 as large amounts of False Positive pixels are removed from sea ice regions, whilst impact on water detection is considered acceptable. For both classifiers, open water detection is comparable to the original classification outputs and is an improvement compared to original `GLCM21` results. Choosing to filter `GLCM11` outputs means we accept the inability to detect the smallest polynyas, as seen in Figure 4.9bV1-2B. Open water detection in other regions is still sufficient (Figure 4.9aV2-2B and 4.9aV4B), while 4.9bV4B shows that sea ice detection of `GLCM11` is improved significantly in this way.

Finally, the filtered classification results are summarized in Figure 5.1a and 5.1b to analyze which detection method of each classifier is preferred. For clarity, we name the filtered outputs of `GLCM5` and `GLCM11` `GLCM5F` and `GLCM11F` respectively. First, we compare each method in regions where polynyas and water leads are present (Figure 5.1a). It is evident that GLCM21 performs very poorly at water edge detection and at thin water lead detection. This is made even more apparent when comparing GLCM21 results directly to GLMC11F and GLCM5F in 5.1aV2-3A and 5.1aV4A. Here the we see that GLCM21 is outperformed by both GLCM11F and GLCM5F as the purple mask is significantly smaller and less accurate than the green and yellow masks of GLCM5F and GLCM11F respectively. GLCM5F outperforms GLCM11F (5.1aV4A, yellow versus green), but not as considerable as it outperforms GLCM21 results (5.1aV4A, yellow and green versus red and purple masks). GLCM5F and GLCM11F results are considered to be roughly equal in terms of small scale open water detection accuracy. Second, we compare detection methods in Figure 5.1b on their ability to accurately detect sea ice, thus minimizing False Positives. Here, we see that regions of False Positives (5.1bV1-1, 5.1bV2, 5.1bV4) are confined to roughly equal locations for every detection method (purple mask). Upon further inspection, these figures also show that GLCM5F, despite an impressive reduction in False Positives, is still most prone to detect sea ice incorrectly as substantial parts of these regions are only detected via this method (yellow mask). We argue that, even though the inherent draw-back of SAR-based classification for water-ice classification is not completely overcome, applying a water body area filter is an effective manner to increase polynya detection performance considerably for GLCM-based classifiers with window sizes up to $w = 11$. Given that GLCM21 output underperforms compared to GLCM11F and GLCM5F for open water detection, and that GLCM5F underperforms compared to GLCM21 and GLCM11F, the proposed method to classify open water bodies for plume-driven polynya detection is to use GLCM11F: classify dual-pol SAR data using GLCM11, then post-process classification results with an water-area filter (>1.2km2).

5.3. Comparison to Other Detection Methods

The method presented in this study is able to identify small scaled water bodies (<440m) with high accuracy for both water-ice classification (kappa = 93%) and water-ice sub-class classification (kappa = 83%). Overall (sub-classification) accuracy is higher than the automated SAR-based river ice detection method presented by de Roda Husman et al. [2021] (kappa = 80%), despite choosing the same window size (*w*=11). Comparing overall accuracy to results from other studies, such as [Hollands and Dierking, 2016] and [Karvonen and Hallikainen, 2009] is more difficult as the authors do not present accuracy metrics. Given that our results are comparable to that of de Roda Husman et al. [2021], we do however believe that the accuracy of our results are on par with results from Hollands and Dierking [2016] and Karvonen and Hallikainen [2009]. It should however be noted, that each of these studies have made more detailed water-ice detection algorithms than our (sub-class) classification algorithm. Hollands and Dierking [2016] have used a decision tree to classify 8 types of ice besides a water class. They however used both L-band and C-band SAR data, as well as brightness temperatures from passive microwave radar imagery to achieve this. As such, temporal resolution of input data is much lower compared to our input data, as images from all three data sets are needed for a classified image. As we want to keep temporal resolution as high as possible, the use of other data sets was omitted, meaning that ability to detect the amount of ice sub-classes presented by Hollands and Dierking [2016] was diminished. Karvonen and Hallikainen [2009] based their detection algorithm on edge-features alone. As expected, their results show high performance for edge detection and show a good ability to classify thin leads of ice. For the scope of this research we only looked at GLCM-based features, and comparing them to those of Karvonen and Hallikainen [2009], we argue that results of GLCM5 (a window size of 5x5 pixels of 40x40m) are comparable to their results, although the method by Karvonen and Hallikainen [2009] does detect the thinnest wate leads more easily. Our method loses performance compared to Karvonen and Hallikainen [2009] when increasing GLCM window size to *w*=11 and above . It could therefore be worthwhile to add edge features to the feature selection procedure in a future study to see whether such features are of added value for mitigating false edge detection (chapter 5).

When comparing classification results of de Roda Husman et al. [2021] to those in this study, we believe that there are two reasons for the improved classification performance. For one, the difference in performance is due to a different choice of classification classes. While class choices are comparable in both studies (de Roda Husman et al. [2021] classify SAR images as *Open Water*, *Rouble Ice* and *Sheet Ice*), the ice sub-classes in this study are more different from a physical standpoint than *Rubble Ice* and *Sheet Ice*, thereby making it easier to classify images correctly. As the main objective of this study is to distinguish water from any type of ice, instead of classifying specific types of



(a) Polynyas and water leads.

(b) Sea Ice.

Figure 5.1: Comparison of polynya detection methods zoomed in at regions of interest.

ice correctly as well, we believe that the class choice used here is valid. Another reason for improved classification could be the slightly different choice of GLCM features. In this study we propose to base the classifier on 'sum average' (HH and HV polarization) and 'difference variance' (HV polarization, whereas de Roda Husman et al. [2021] proposes the use of original cross-polarization backscatter intensity (VH in their case), the co-polarization GLCM average (VV) and a polarimetric based feature (not used in this study). Evidently, feature choice between both studies is relatively similar `HH_savg` relates to VV GLCM average, while `HV_savg`, given its high correlation with `HV` (94.8%), is comparable to VH. de Roda Husman et al. [2021] used a polarimetric feature to improve water-river ice distinction, where we show that a GLCM variance feature based on cross-polarized data leads to similar results for (small-scaled) water-sea ice detection.

Lohse et al. [2021] have proposed another feature choice for sea ice classification. They proposed the use of original dual-pol intensities and co-polarization Dissimilarity, Contrast and Energy (features 14, 2 and 1 in Table 2.2 respectively). They do not use HV-based features, stating that HV is more problematic than HH as signals are often close to the noise floor. Our results prove that that in the context of small-scaled water detection, cross-polarization data is very useful, when necessary pre-processing steps such as thermal noise correction are applied [Lohse et al., 2021]. Despite a different detection goal, namely detecting different types of sea ice instead of small-scaled water detection, the feature choices themselves are again similar to those in this study. Dissimilarity and Contrast are highly correlated to variance (99.5% and 97.0% respectively for HV) and will have similar spectral characteristics as `HV_dvar`. The difference in feature choice is between both studies is characterized via the use of Energy, as `HV_dvar` has little correlation with `HV_asm` (15.2%). Energy is therefore more likely a useful feature for the distinction of different ice types, than for the separation of water and sea ice.

From a spatial context, our results show that window sizes larger than *w*=11 should be avoided. This is in contrast with results presented by Lohse et al. [2021], who state that water-ice classification accuracy is improved by choosing a window size up to *w*=51 (for a 40x40m pixel size). In a general sense, our results are in accordance with these findings (Table 4.1). `GLCM21` led to the highest overall accuracy, but the loss in spatial resolution resulted in a poorer ability to detect water features on a spatial scale of small plume-driven polynyas. Lohse et al. [2021] have mentioned as well that the loss of spatial resolution is a consequence for the increase in window size, although it did not have a negative impact on their results, given that spatial resolution could be relatively low for their study goal. For the accurate detection of small-scaled water bodies however, this is not the case, which is why *w*=11 is advised for plume-driven polyna detection.

One of the weaknesses of the method presented in this study is the ability to accurately detect plume-driven polynyas from other bodies of water. The amount of falsely classified small-scaled water bodies is, at the moment, too significant to be able to use an area threshold for plume-driven polynya demarcation. Studies from Alley et al. [2016, 2019], Gladish et al. [2012]; Lazeroms et al. [2018]; Sergienko [2013], Shean et al. [2019] have had, in this regard, more success, albeit for different reasons. For one, Alley et al. [2016, 2019] have identified plume-driven polynyas manually from MODIS data. The fact that their method is not automated and based on optical imagery means that it is more time consuming and poorer in temporal coverage than the method presented here. Gladish et al. [2012]; Lazeroms et al. [2018]; Sergienko [2013] have modelled basal melt channels directly using ocean models. The major drawback of their approach is the significant computational effort and expertise needed to compute basal melt patterns and consequently predict melt channels and possible plume-driven polynya locations. The method we present here has extremely low computational effort, given its cloud-based nature, and can be applied by any GEE user. There are other methods to predict basal melt, such as using high resolution DEMs to infer Ice Shelf bottom topography [Shean et al., 2019]. However, for an accurate prediction of basal melt channels commercial data was used with high enough spatial resolution. This means that high temporal resolution is consequently lower, given that these campaigns are costly and time consuming. For the mapping of basal melt channels, lower temporal resolution is acceptable, but for an accurate overview of spatio-temporal coverage of plume-driven polynyas this is not sufficient.

Overall, despite having poorer applicability for polynya detection than other polynya or basal melt detection methods, there is potential to make detection significantly faster, easier and more accessible than the current methods available, We therefore urge to build upon this SAR-based detection method, until results are satisfactory for a fully automated plume-driven polynya detection algorithm. Based on our results and comparisons to other research we argue that the lack in accuracy can be overcome by adding some extra post-processing steps, or aiding the classifier with extra SAR-based information, which we will discuss further in the section below.

5.4. Recommendations

Even though `GLCM11F` achieves very high accuracy in the detection of water bodies, it does not yet provide information on what body of water would classify as a plume-driven polynya. A direct way to try and gain this knowledge from `GLCM11F` results can be to apply a second area filter, but now with an upper threshold (<25km2 for instance). However, as we can see by comparing results from Figure 5.1b, this will still leave many regions of False Positive in the eventual 'plume polynya' detection. This means some other form of post-processing has to be applied to infer this information more accurately from the `GLCM11F` classification output. Given the physical characteristics of plume-driven polynyas (chapter 1), it would be interesting to classify dual-pol SAR data using `GLCM11F` in a specific region of interest, such as the Pine Island Glacier Embayment, for a multi-year period. Then, we suggest to create a pixel-based water detection frequency map of the classification output to see what the spatio-temporal distribution of open water detection is like. As we expect that plume-driven polynyas are confined to end points of basal melt channels, these regions should therefore be relatively consistently detected as open water on roughly the same location, resulting in high frequency values. False Positives in sea ice are a result of a process that is much more dependent local meteoroligcal conditions and on the local angle of incidence. This type of water detection is therefore expected to be more irregular (both spatially and temporally). We expect that these detected water bodies have much lower frequency values on could be filtered out by applying a relatively high frequency threshold. This way the detection has the potential to be truly optimized for plume-driven polynya detection.

Another proposed method to extract extra information from dual-pol SAR data to aid polynya-specific detection is by making use of a Conventional Neural Network (CNN). A CNN is specially designed to extract spatial patterns from the input imagery and application on SAR imagery has the potential to directly classify bodies of water as polynyas and water leads. As polynyas and leads are distinctly shaped and have much different spatial features than smooth sea ice (see Figure 4.3), we assume that a CNN can significantly increase classification reliability and accuracy, whilst directly focusing on polynya detection instead of open water detection.

Even though `GLCM11F` is not yet accurate enough for automated plume-driven polynya detection, the classifier can still serve a useful purpose in its current state. We recommend to evaluate this classifier as a tool to validate basal melt predictions from DEMs and ocean and ice models, as its ability to detect (small scale) bodies of water accurately is sufficient, even though it is not able to directly distinguish plume-driven polynyas from other bodies of water.

6 Conclusion

Using specific texture metrics to improve the detection of water in polar environments leads to promising results. We created a GLCM-based classifier that facilitates the detection of small scaled (1-10km2) open water bodies from dual-pol SAR imagery. The distinction between rougher water states and smoother sea ice states is significantly improved compared to a dual-pol SAR classifier, showing the added value of using GLCM features to detect plume-driven polynas.

We tested four different classifiers on their ability to classify water and ice. The first classifier was based on the original dual-pol SAR imagery (backscatter intensity in HH and HV polarization) and was used as a base classifier to compare the other classifiers to. We computed GLCM features of various window sizes (*w*=[5,11,21]) and assessed how classifiers for each distinct GLCM feature data set performed. Then, we post-processed the GLCM-based classification outputs in an effort to remove incorrectly classified water regions and to improve the identification of plume-driven polynyas from other open water bodies. In doing so, we tried to answer the question:

What are the strengths and limitations of classifying and post-processing dual-pol SAR data and GLCM features for the detection of plume-driven polynyas?

Below, we go through each sub-question to provide an answer.

(1) *How is open water classification affected when basing a classifier on dual-pol GLCM features, compared to classification where only dual-pol SAR data is used?*

(a) *Which GLCM features based on HH or HV polarized SAR data prove most useful for water detection?*

`HH` and `HV` perform reasonably well when looking at accuracy score in `BASE`, but lead to too many False Positives and False Negatives (Figure 4.7) for reliable use in polynya detection. The best features, chosen from a selective process based of feature importance scores and feature correlations, are `HH_savg`, `HV_savg` and `HV_dvar`. The choice of these features is independent of window size choice, having highest uncorrelated importance scores for each GLCM-based classifier. We conclude that the application of GLCM features aids in more accurately distinguishing water from ice. GLCM features pick up spatial context around an original `HH` and `HV` pixel. In this way, outliers and irregular intensity patterns are smoothed out from `HH` and `HV` data at the expense of reduced spatial resolution. Where classification output of `BASE` shows many False Positives and False Negatives, each of the GLCM-based classifiers are much more robust and show more spatially uniform classification outputs, in accordance with optical reference data. The added value of `HH_savg` and `HV_savg` compared to `HH` and `HV` lies in this smoothing effect, thereby reducing the classifier's sensitivity to noise. When looking into feature intensity distributions per (sub-)class, we conclude that `HH_savg` is a useful metric to separate *Rough Water* from *Sea Ice* as overlap between the two subclass is small (Figure 4.2). `HV_savg` shows a poorer ability to distinguish these sub-classes. For this, `HV_dvar` provides additional useful information, as Figure 4.2 shows that *Water-Ice* distinction is aided by including this feature. Overlap between the (*Rough*) *Water* and (*Sea*) *Ice* is minimal and we conclude that classification without `HV_dvar` leads to less accurate water body detection (see also Figure 4.7).

(b) *How does the GLCM window size affect classification performance of water bodies?*

Every GLCM-based classifier shows significant improvements compared to `BASE`, where increasing *w* leads to higher classification accuracy (`BASE`: kappa = 84.4%, `GLCM5`: kappa = 91.5%, `GLCM11`: kappa = 92.7%, `GLCM21`: kappa = 93.0%). FNR and FPR of *BASE* are a factor 2-3 higher than for the GLCM-based classifiers. For the GLCM-based classifiers, FNR decreases consistently with increased *w* (`GLCM5`: FNR = 4.74%, `GLCM11`: FNR = 4.49%, `GLCM21`: FNR = 4.25%), while showing equal omission errors for classification of *Rough Water* as *Sea Ice* (10%). The FPR of `GLCM5` is higher than `GLCM11` and `GLCM21` which show comparable results (`GLCM5`: FPR = 3.72%, `GLCM11`: kappa = 2.83%, `GLCM21`: kappa = 2.78%), which arises from a lower ability of `GLCM5` compared to `GLCM11` and `GLCM21` to detect *Sea Ice* as *Rough Water* (ommission errors are 8%, 5% and 5% respectively). Increasing window size to $w = 21$ is thus optimal from a data analysis standpoint, with $w = 11$ showing nearly equal performance. The choice of $w = 21$ however leads to a significant decrease in spatial resolution: from 40m to 840m. For this reason, spatial interpretation shows that influence of window size is in reality more subtle. While increasing *w* increases overall accuracy, it also reduces spatial resolution of the classifier. Because of this, the ability of `GLCM21` to detect smaller scaled (<1.2km2) bodies of water is significantly poorer than `BASE` or `GLCM5`. Besides an inability to detect such small scaled spatial features, the application of GLCM-based features also introduces reduced performance in water-ice-edge detection. We conclude that every GLCM-based classifier misinterprets such water edges and introduces so-called False Ice Edge detection. The thickness of these False Ice Edges is directly proportional to the classifier's window size *w*. In this view, `GLCM5` significantly outperforms `GLCM21` due to its roughly four times higher spatial resolution. As expected, `GLCM11` shows performance that is in between both these classifiers. We conclude that the (small scaled) open water body detection of `GLCM5` and `GLCM11` is sufficient for polynya detection, but consider `GLCM21` performance inadequate. Conversely, for accurate classification of (*Sea*) *Ice*, increasing *w* reduces the number of False Positives in the classification output. When comparing `GLCM5` to `GLCM21`, results show that `GLCM5` leads to significantly more False Positives than `GLCM21`. Reducing spatial resolution in essence results in a filtering step during classification of smaller regions of sea ice that would otherwise have been detected as *Water*. `GLCM21` is able to minimize False Positives considerably, but for this we consider the performance `GLCM5` to be inadequate. `GLCM11` classification results in a significant amount of False Positives as well. The extent is considerably less than the `GLCM5` output, but we conclude that `GLCM11` is very much prone to misinterpret *Sea Ice* as *Water*.

The size of *w* therefore is a balance between minimizing False Positives (high *w*) and minimizing False Negatives (low *w*). Based on comparisons of open water detection and sea ice detection we conclude that, without any forms of post-processing, a window size of $w = 11$ is the best choice for (small scale) open water detection.

(2) In what way do post-processing steps improve plume-driven polynya detection from ice-water classified images?

In this study, we have assessed the use of post-processing classification outputs by applying an area filter on classified bodies of water (>1.2km2). The threshold of 1.2km2 was chosen based on minimum areas of plume-drivenn polynya observations. This additional filter step proved particularly useful for `GLCM5`, as most classified water bodies below this size were False Positives in sea ice regions and removal of True Positives was minimal. The filter step also proved beneficial for `GLCM11`, as many False Positives on sea ice were below this threshold as well. However, filtering did negatively impact the ability of *GLCM11* to detect the smallest of water bodies ($\sim$ 1-2km2). Filtering `GLCM21` did not prove to be particularly useful, as most classified bodies of water for this classifier were already larger than 1.2km2. Filtering for bodies smaller than this threshold removes some True Positives from the classification output, which is already not a strong point of this classifier. In the end, we conclude that post-processing `GLCM5` and `GLCM11` in this manner is beneficial for plume-driven polynya detection (called `GLCM5F` and `GLCM11F` respectively). For `GLCM21`, polynya detection is not substantially improved and so we conclude that filtering `GLCM21` is not advised. Even though significant numbers of False Positives are removed from the original classification outputs in `GLCM5F` and `GLCM11F`, we still encounter too many False Positives in the filtered result to automatically detect plume-driven polynyas. Both detection methods have their own strengths and weaknesses. `GLCM5F` proves useful for highly accurate polynya detection but, while minimized, False Positives are prevalent. `GLCM11F` minimizes these False Positives much more efficiently, while being able to classify smaller scaled polynyas, although not down to less than roughly 2km2. Both methods have the potential to detect plume-driven polynyas with even higher accuracy in an automatic fashion. However, extra post-processing steps are needed to process these results further in order to achieve this goal. Applying an upper limit could help in distinguishing plume polynyas from larger other water bodies, but this process is still prone to leave in False Positives.

We believe that an extra post-processing step on filtered results via a temporal analysis can help to further filter out these False Positives. Plume-driven polynyas are expected to be spatially confined to end points of basal melt channels, expectedly leading to relatively consistent detections of water in such regions. False Positives in sea ice regions are the result surface characteristics that are more dependent on local meteorological and seasonal conditions, expectedly leading to significantly less temporally consistent detection of water. In this way, a temporal filter can be applied to remove False Positives from classification outputs.

To conclude, our results have shown that using texture based dual-pol classifiers improves classification significantly compared to dual-pol only classifiers. A classifier based on GLCM features with a window size *w* of 11x11 pixels (440x440m) proved to be most successful in accurate water body detection and sea ice detection. Using an area filter as a post-processing method to improve plume-driven polynya detection accuracy proved successful when a classifier is based on GLCM data with window sizes no larger than 11x11 pixels (440x440m). Although successful in improving polynya detection, False Positives are still prevalent in the filtered classification outputs and additional post-processing steps, such as a temporal analysis, are advised for automatic and unsupervised plume-driven polynya detection, in order to overcome the inherent difficulty of using SAR data for water-ice classification. Nevertheless, in its current form, this method could be used to validate predicted locations of basal melt.

A
Tables & Figures

Table A.1: Image IDs of each Senintel-1 and LandSat-8 match.

Match ID	Data Set	Image ID
T1	Sentinel-1	1_S1A_EW_GRDM_1SDH_20171227T045224_20171227T045328_019883_021D66_029C
	LandSat-8	1_LC08_161131_20171227
	LandSat-8	1_LC08_161132_20171227
T2	Sentinel-1	1_S1A_EW_GRDM_1SDH_20171222T044413_20171222T044517_019810_021B27_D74B
	LandSat-8	1_LC08_158131_20171222
T3	Sentinel-1	1_S1A_EW_GRDM_1SDH_20181219T042800_20181219T042904_025089_02C4EA_6C2C
	LandSat-8	1_LC08_164131_20181219
T4	Sentinel-1	1_S1A_EW_GRDM_1SDH_20211227T042817_20211227T042922_041189_04E500_4D0F
	LandSat-8	1_LC08_164131_20211227
T5	Sentinel-1	1_S1A_EW_GRDM_1SDH_20180112T041941_20180112T042056_020116_0224CF_DCC6
	LandSat-8	1_LC08_161131_20180112
T6	Sentinel-1	1_S1A_EW_GRDM_1SDH_20181214T041948_20181214T042052_025016_02C249_1D9D
	LandSat-8	1_LC08_161132_20181214
T7	Sentinel-1	2_S1A_EW_GRDM_1SDH_20171214T072820_20171214T072853_019695_02178E_F273
	LandSat-8	1_LC08_182131_20171214
T8	Sentinel-1	2_S1A_EW_GRDM_1SDH_20180108T063108_20180108T063213_020059_0222EC_D38D
	LandSat-8	1_LC08_181131_20180108
T9	Sentinel-1	2_S1A_EW_GRDM_1SDH_20180109T053339_20180109T053443_020073_02236D_FA92
	LandSat-8	1_LC08_172132_20180109
V1	Sentinel-1	1_S1A_EW_GRDM_1SDH_20171220T050036_20171220T050140_019781_021A4B_C390
	LandSat-8	1_LC08_160131_20171220
V2	Sentinel-1	1_S1A_EW_GRDM_1SDH_20171224T042753_20171224T042857_019839_021C01_4F12
	LandSat-8	1_LC08_156131_20171224
	LandSat-8	1_LC08_156132_20171224
V3	Sentinel-1	1_S1A_EW_GRDM_1SDH_20200107T042805_20200107T042910_030689_03848D_515B
	LandSat-8	1_LC08_164131_2020010
V4	Sentinel-1	2_S1A_EW_GRDM_1SDH_20171219T055833_20171219T055937_019767_0219E2_C668
	LandSat-8	1_LC08_169132_20171219

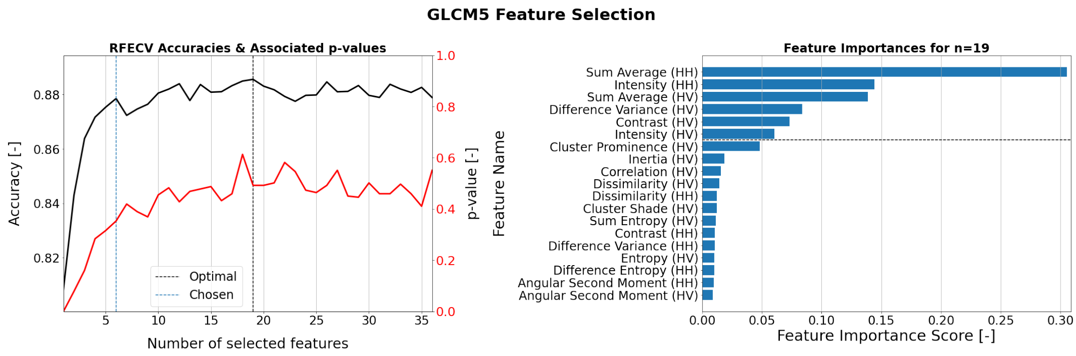


Figure A.1: Feature Selection for the `GLCM5` classifier. Left: In blue, recursive Feature Elimination with Cross Validation to select the optimal number of features and in red a student-t test to assess statistical independence for each number of features. Right: feature importance for the chosen number of features (n=6).

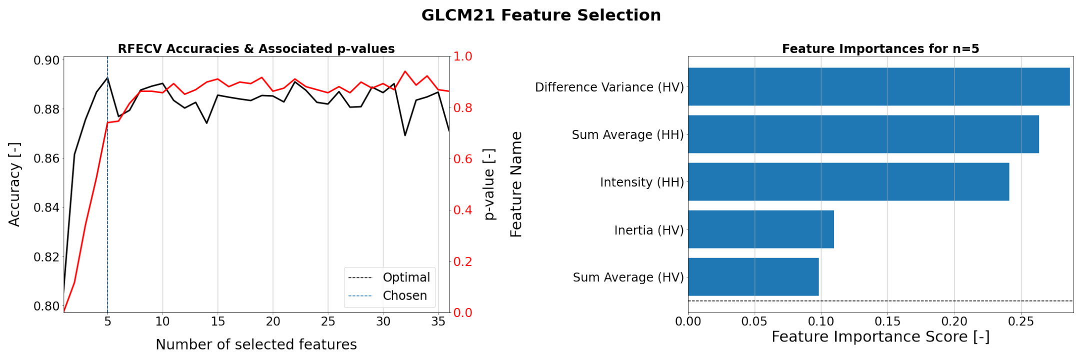


Figure A.2: Feature Selection for the `GLCM21` classifier. Left: In blue, recursive Feature Elimination with Cross Validation to select the optimal number of features and in red a student-t test to assess statistical independence for each number of features. Right: feature importance for the chosen number of features (n=5).

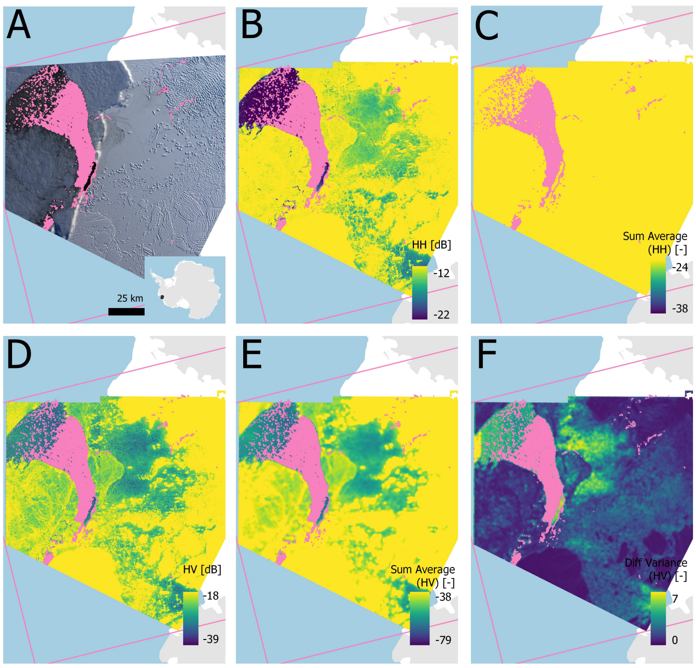


Figure A.3: `HH`, `HV` and `GLCM21` texture features for *V1*. Optical image is added for visual reference. For clarity, we highlighted water extent in the optical image by overlaying its water-ice border on top of every image in pink.

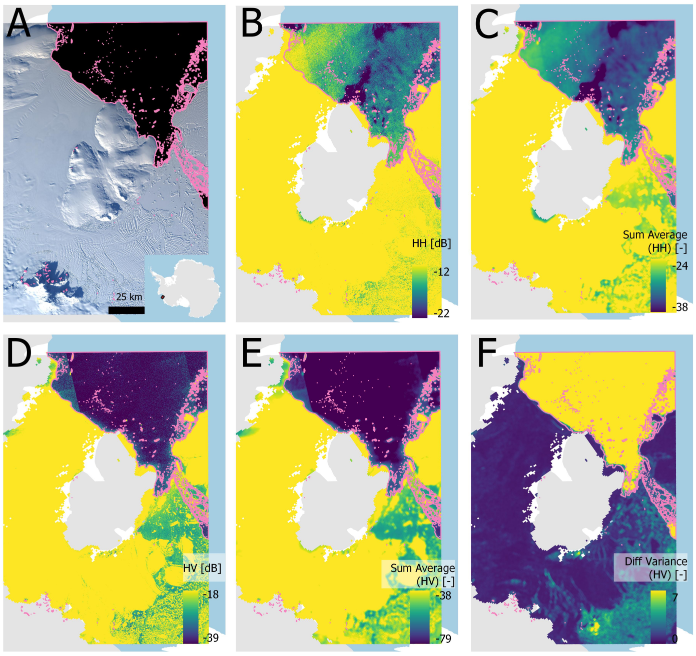


Figure A.4: `HH`, `HV` and `GLCM21` texture features for *V3*. Optical image is added for visual reference. For clarity, we highlighted water extent in the optical image by overlaying its water-ice border on top of every image in pink.

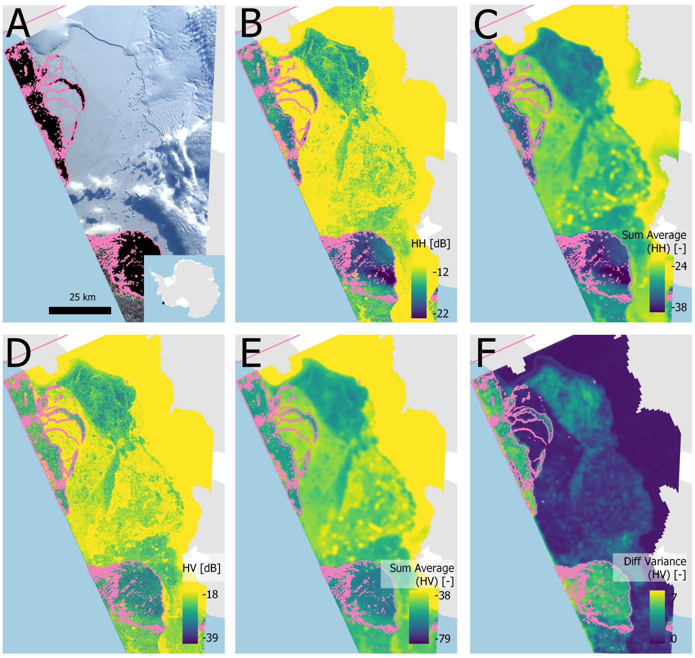


Figure A.5: `HH`, `HV` and `GLCM21` texture features for *V4*. Optical image is added for visual reference. For clarity, we highlighted water extent in the optical image by overlaying its water-ice border on top of every image in pink.

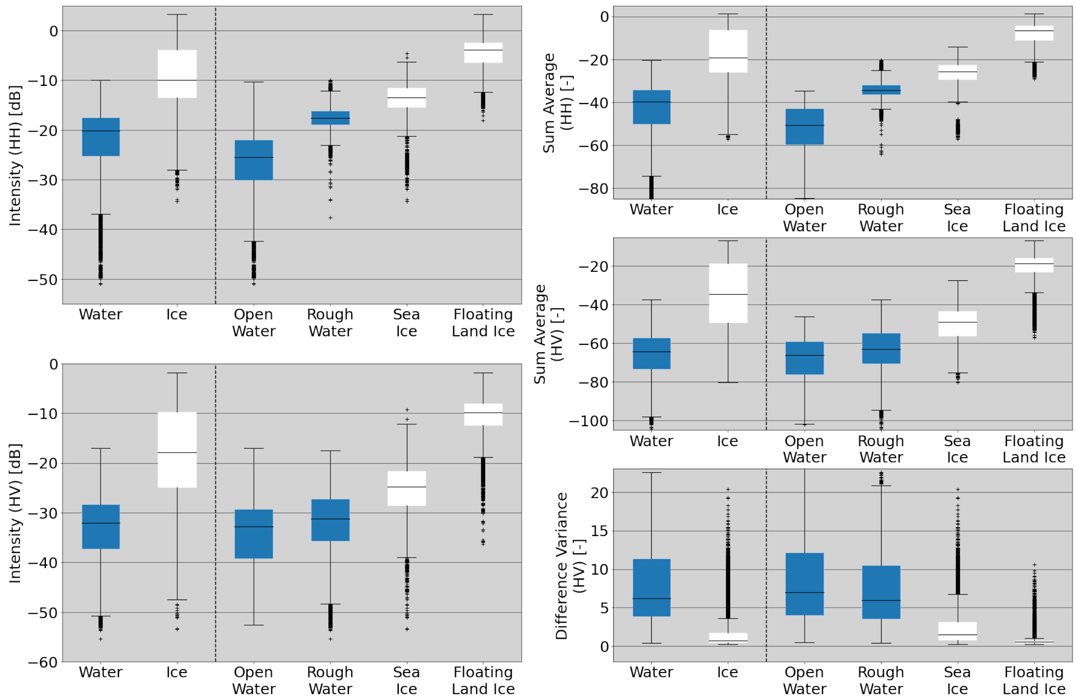


Figure A.6: Boxplots showing the distribution of backscatter intensities (left) and textural features (right) for `GLCM5` training data

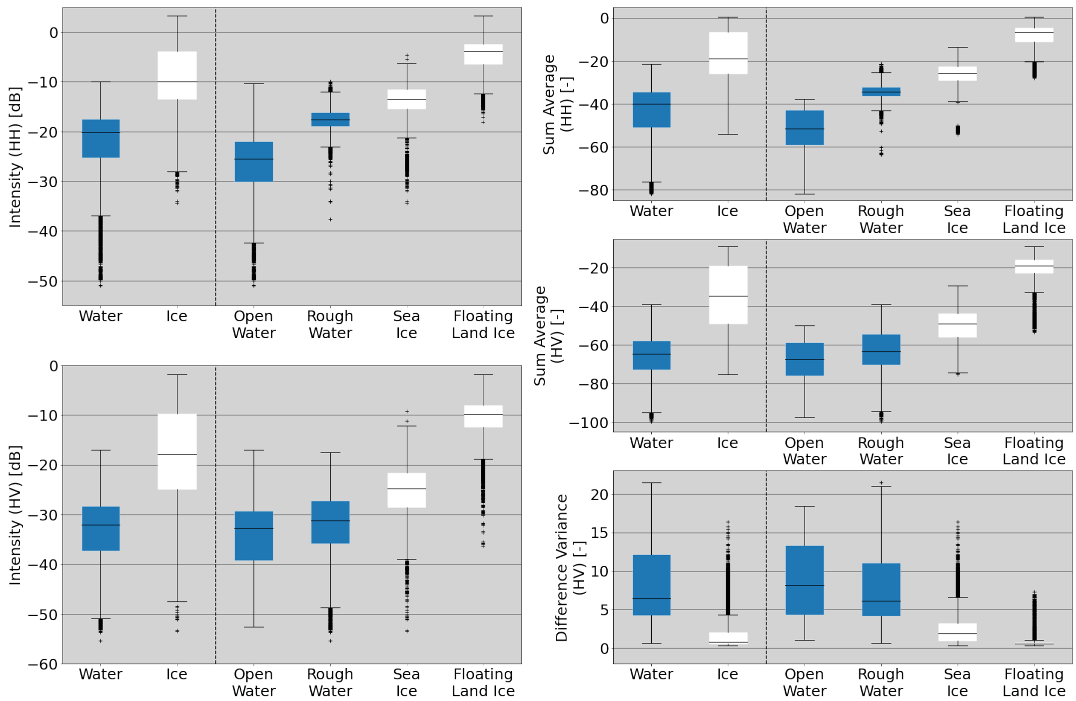
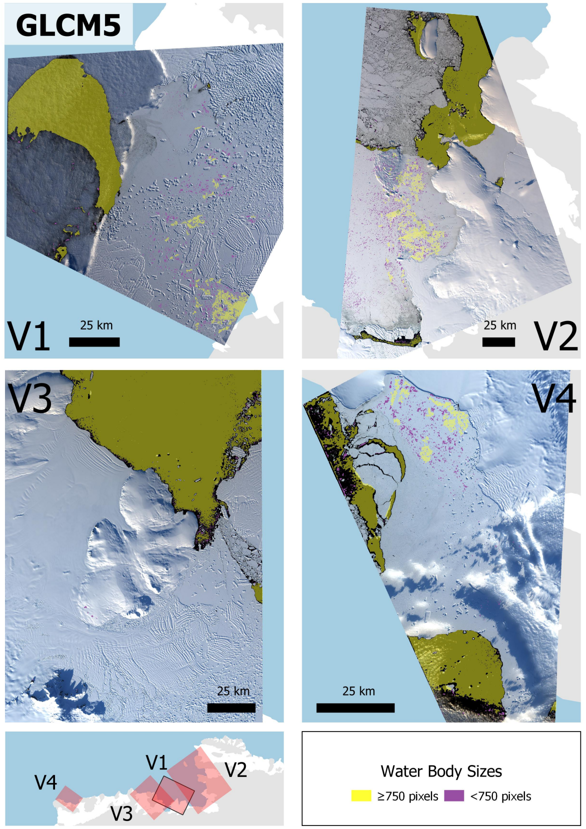
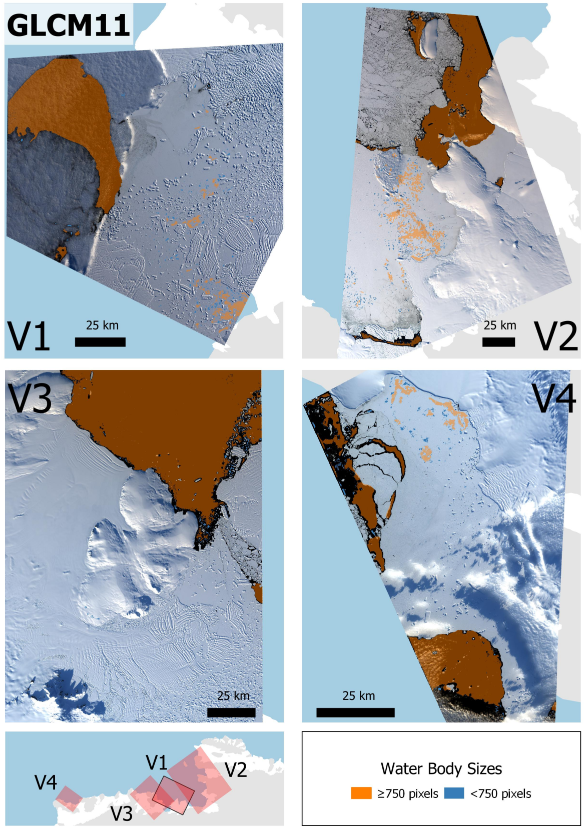


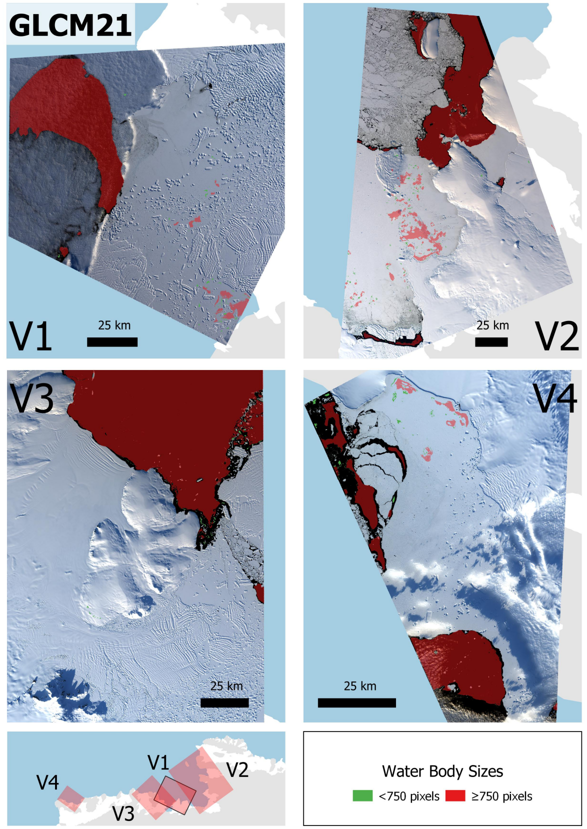
Figure A.7: Boxplots showing the distribution of backscatter intensities (left) and textural features (right) for `GLCM21` training data



(a) Effect of applying an area filter on `GLCM5` classification output.



(b) Effect of applying an area filter on `GLCM11` classification output.



(c) Effect of applying an area filter on `GLCM21` classification output.

Figure A.8: Post-processing filtering step where bodies of water larger than 750 pixels are disintinguished from those smaller than 750 pixels.

B

Code

B.1. Ice Shelf Coastlines

1_IceShelfCoastLines

```
#%% (1) Code set-up
## (1.1) Import packages
import ee
import geemap

## (1.2) Initialize
ee.Initialize()

#%% (2) Create Ice Shelf Coast Line Geometry
## Leave commented!!
## Export was necessary to create feature collections used in section below

# IceShelves = ee.FeatureCollection("users/sophiederoda/IceShelf_Antarctica")
# GroundingLine = ee.FeatureCollection("users/sophiederoda/GroundingLine_Antarctica")
# Union = IceShelves.geometry(1e3).union(GroundingLine.geometry(1e3),maxError=1e3)
# UnionPoly = ee.Geometry(Union.geometries().get(-1))
# CoastLineGeom = ee.Geometry.LineString(ee.List(UnionPoly.coordinates().get(0)))
# CoastLine = ee.FeatureCollection([ee.Feature(CoastLineGeom)])

# geemap.ee_export_vector_to_drive(
#     ee_object = CoastLine,
#     description = 'AntarcticCoastLine',
#     folder = 'CoastLine',
#     file_format = 'shp')

IceShelves = ee.FeatureCollection("users/sophiederoda/IceShelf_Antarctica")
CoastLine = ee.FeatureCollection('users/skjeltmaps/AntarcticCoastLine')


## (2.1) Divide Coast Line Geometry in Segments
def defineSubCoastLines(IceShelfFeature,CoastLineFeatureCollection,maxDistance):
    IS = IceShelfFeature
    ISgeom = IS.geometry(1e3)
    IScoords = ee.List(ISgeom.coordinates().get(0))
    ISpoly = ee.Geometry.Polygon(IScoords)
    ISbuffer = ISpoly.buffer(1e2)
    ISfeat = ee.Feature(ISbuffer).copyProperties(IS)

    CL = CoastLineFeatureCollection.first()
    CLgeom = CL.geometry(1e3)
    CLcoords = ee.List(CLgeom.coordinates().get(0))
    CLmultipoint = ee.Geometry.MultiPoint(CLcoords)

    INTgeom = CLmultipoint.intersection(ISbuffer,maxError=5e3)
    INTfeat = ee.Feature(INTgeom)

```

```
    letters = ee.List(['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R
        ','S','T','U','V','W','X','Y','Z'])
    INTcoords = INTfeat.geometry(1e3).coordinates()
    INTindeces = INTcoords.map(lambda coord: CLcoords.indexOf(ee.List(coord))).removeAll(ee.
        List([-1]))
    ISCLcoords = ee.List.sort(INTindeces).map(lambda index: ee.List(CLcoords.get(index)))
    westbound = ee.List(ISCLcoords.get(0))
    ITERlist = ee.List([ee.List([0]),westbound,ISCLcoords,maxDistance,ee.List([0])])

    def findInterruptions(coordsIterate,iterations):
        newCoords = ee.List(coordsIterate)
        indexList = ee.List(ee.List(iterations).get(0))
        oldCoords = ee.List(ee.List(iterations).get(1))
        coordinatesList = ee.List(ee.List(iterations).get(2))
        maxDistance = ee.Number(ee.List(iterations).get(3))
        distanceList = ee.List(ee.List(iterations).get(4))
        distance = ee.Geometry.Point(oldCoords).distance(ee.Geometry.Point(newCoords),
            maxError=1e3)

        return ee.Algorithms.If(
            condition = distance.gt(maxDistance),
            trueCase = ee.List([\
                ee.List(indexList.add(ee.Number(coordinatesList.indexOf(newCoords)))),
                ee.List(newCoords),
                ee.List(coordinatesList),
                ee.Number(maxDistance),
                ee.List(distanceList).add(ee.Number(distance))]),
            falseCase = ee.List([\
                ee.List(indexList),
                ee.List(newCoords),
                ee.List(coordinatesList),
                ee.Number(maxDistance),
                ee.List(distanceList).add(ee.Number(distance))]))

    SUBCLindeces = ee.List(ee.List(        ISCLcoords.iterate(
            function = findInterruptions,
            first = ITERlist)).get(0)).add(-1)

    def sliceSubCoastLines(coords,interruptions,feature):
        numberOfInterruptions = interruptions.size()
        numberOfSlices = numberOfInterruptions.subtract(1)
        indexList = ee.List.sequence(0,numberOfSlices.subtract(1))
        slicedCoords = indexList.map(lambda i: ee.List(coords.slice(
            ee.Number(interruptions.get(ee.Number(i))),
            ee.Number(interruptions.get(ee.Number(i).add(1)))
        )))
        slicedFeatures = slicedCoords.map(lambda coords: ee.Feature(ee.Geometry.LineString(ee
            .List(coords))).copyProperties(feature))
        name = ee.String(feature.get('NAME'))

        return ee.Algorithms.If(
            condition = numberOfSlices.gt(1),
            trueCase = indexList.map(lambda i: ee.Feature(slicedFeatures.get(i))\
                .set('NAME',name.cat(ee.String('_')).cat(ee.String(letters.get(i))))),
            falseCase = slicedFeatures)

    ISCLlist = ee.List(sliceSubCoastLines(ee.List(ISCLcoords),ee.List(SUBCLindeces),ee.
        Feature(ISfeat)))
    ISCL = ee.FeatureCollection(ISCLlist)

    return ISCL

def defineSegmentedCoastLines(subCoastCol,maxDistance):
    def segmentCoastLines(subCoastFeature,maxDistance):
        feature = ee.Feature(subCoastFeature)
        coords = feature.geometry().coordinates()
        westbound = ee.List(coords.get(0))
        eastbound = ee.List(coords.get(-1))
        iterationlist = ee.List([ee.List([0]),westbound,coords,maxDistance,ee.List([0])])

        def findSegments(coordsIterate,iterations):
```

```
            newCoords = ee.List(coordsIterate)
            indexList = ee.List(ee.List(iterations).get(0))
            oldCoords = ee.List(ee.List(iterations).get(1))
            coordList = ee.List(ee.List(iterations).get(2))
            maxDistance = ee.Number(ee.List(iterations).get(3))
            distanceList = ee.List(ee.List(iterations).get(4))
            distance = ee.Geometry.Point(oldCoords).distance(ee.Geometry.Point(newCoords), maxError=1e3)
            cumdist = ee.Number(distanceList.reduce(ee.Reducer.sum())).add(ee.Number(distance))

            return ee.Algorithms.If(
                condition = cumdist.gt(maxDistance),
                trueCase = ee.List([\
                    ee.List(indexList.add(ee.Number(coordList.indexOf(newCoords)))),
                    ee.List(newCoords),
                    ee.List(coordList),
                    ee.Number(maxDistance),
                    ee.List([0])]),
                falseCase = ee.List([\
                    ee.List(indexList),
                    ee.List(newCoords),
                    ee.List(coordList),
                    ee.Number(maxDistance),
                    ee.List(distanceList).add(ee.Number(distance))]))

        indeces = ee.List(ee.List(coords.iterate(
                function = findSegments,
                first = iterationlist)).get(0))

        segmentIndeces = ee.Algorithms.If(
            condition = ee.Number(indeces.get(-1)).eq(ee.Number(coords.indexOf(eastbound))),
            trueCase = indeces,
            falseCase = indeces.add(coords.indexOf(eastbound)))

        numberOfSegments = ee.List(segmentIndeces).size()
        numberOfSlices = numberOfSegments.subtract(1)
        indexList = ee.List.sequence(0,numberOfSlices.subtract(1))
        slicedCoords = indexList.map(lambda i: ee.Algorithms.If(
            condition = ee.Number(ee.List(segmentIndeces).get(ee.Number(i).add(1))).eq(ee.Number(coords.indexOf(eastbound))),
            trueCase = ee.List(coords.slice(
                ee.Number(ee.List(segmentIndeces).get(ee.Number(i))),
                ee.Number(ee.List(segmentIndeces).get(ee.Number(i).add(1))))).add(eastbound),
            falseCase = ee.List(coords.slice(
                ee.Number(ee.List(segmentIndeces).get(ee.Number(i))),
                ee.Number(ee.List(segmentIndeces).get(ee.Number(i).add(1)))))
        ))

        segmentlist = indexList.map(lambda i: ee.Feature(ee.Geometry.LineString(ee.List(slicedCoords.get(i))))            .copyProperties(feature)            .set('SEG', ee.String(ee.Number(i))))

        featCol = ee.FeatureCollection(segmentlist)

        return featCol

    segmentFeatCol = ee.FeatureCollection(subCoastCol).map(lambda feature: segmentCoastLines(feature,maxDistance))

    return segmentFeatCol

## Segment distance was set to 20km
maxDistance = 20e3
subCoastLines = IceShelves.map(lambda iceshelf: defineSubCoastLines(iceshelf,CoastLine,maxDistance)).flatten()
segCoastLines = defineSegmentedCoastLines(subCoastLines,maxDistance).flatten()

## Leave commented!!
## Export was necessary to create feature collections used in section below

```

```
# geemap.ee_export_vector_to_drive(
#     ee_object = segCoastLines,
#     description = 'ISCLSegments'+str(int(maxDistance/1e3))+'km',
#     folder = 'Geometries\IceShelfCoastLineSegments',
#     file_format = 'shp')
```

B.2. (Reference) Image Metadata

2_Metadata

```
#%% (1) Code set-up
## (1.1) Import packages
import ee
import geemap

## (1.2) Initialize
ee.Initialize()

## (1.3) Parameterization
start = '2014-01-01'
end = '2022-04-01'
cloudcover = 20
whiteness = 3e4
S1band = 'HV'
dt = 4

L8bands = ['B4','B3','B2']
S2bands = ['B4','B3','B2']
RGBbands = ['Red','Green','Blue']

IceShelves = ee.FeatureCollection("users/sophiederoda/IceShelf_Antarctica")
ROIs = ee.FeatureCollection('users/skjeltmaps/ISCLSegments20km')

#%% (2) Optical Data Set
## (2.1) Cloud Mask Functions
def cloudMaskL8(image):
    def getQABits(image, bitStart, bitEnd, name):
        pattern = 0
        for i in range(bitStart,bitEnd+1):
            pattern += 2**i

        return ee.Image(image).select([0], [name]).bitwiseAnd(pattern).rightShift(bitStart)

    QA = ee.Image(image).select('BQA').toInt()
    cloud = getQABits(QA,4,4,'CLOUD')
    cloud_confidence = getQABits(QA,5,6,'CLOUD_CONFIDENCE')
    cloudshadow_confidence = getQABits(QA,7,8,'CLOUDSHADOW_CONFIDENCE')
    cirrus_confidence = getQABits(QA,11,12,'CIRRUS_CONFIDENCE')

    return ee.Image(image)
        .updateMask(cloud.eq(0))
        .updateMask(cloud_confidence.lt(3))
        .updateMask(cloudshadow_confidence.lt(3))
        .updateMask(cirrus_confidence.lt(3))

def cloudMaskS2(image):
    qa = ee.Image(image).select('QA60').toInt()

    cloudBitMask = 1 << 10
    cirrusBitMask = 1 << 11

    cloud = qa.bitwiseAnd(cloudBitMask).eq(0)
    cirrus = qa.bitwiseAnd(cirrusBitMask).eq(0)

    return ee.Image(image)
        .updateMask(cloud)
        .updateMask(cirrus)

def addRGBbands(image,bandnames):
    renamed = image.select(bandnames,RGBbands)
```

```
    rgb = image.addBands(renamed).select(RGBbands)
    return rgb

def addMaxWhiteness(image):
    maxvalues = image.select(['B2','B3','B4']).reduceRegion(
        reducer = ee.Reducer.max(),
        bestEffort = True,
        maxPixels = 1e3)
    blue = ee.Number(maxvalues.get('B2'))
    green = ee.Number(maxvalues.get('B3'))
    red = ee.Number(maxvalues.get('B4'))
    total = blue.add(green).add(red)
    return image.set('mean',total)
## (2.2) Preprocessing & merging
def mergeOpticalData(ROIfeat,start,end,cloudcover,whiteness):
        #--------------------  General  --------------------#
        roi = ROIfeat.geometry()

        #-------------------- Landsat 8 --------------------#
        L8 = ee.ImageCollection("LANDSAT/LC08/C01/T2")
            .filterDate(start,end)
            .filterBounds(roi)
            .filterMetadata('CLOUD_COVER', 'less_than', cloudcover)
            .map(lambda image: image.set('Data','L8'))
            .map(addMaxWhiteness)
            .filterMetadata('mean', 'greater_than', whiteness)
            .sort('system:time_start')

        #-------------------- Sentinel 2 --------------------#
        S2 = ee.ImageCollection("COPERNICUS/S2")
            .filterDate(start, end)
            .filterBounds(roi)
            .filterMetadata('CLOUDY_PIXEL_PERCENTAGE', 'less_than', cloudcover)
            .map(lambda image: image.set('Data','S2'))
            .map(addMaxWhiteness)
            .filterMetadata('mean', 'greater_than', whiteness)
            .sort('system:time_start')

        #--------------------   Merged   --------------------#
        Opt = L8.merge(S2).sort('system:time_start')

        return ee.List([L8, S2, Opt])

#%% (3) Sentinel-1 Data Set
def preProcessS1(ROIfeat,S1band,start,end):
        #--------------------    General    --------------------#
        roi = ROIfeat.geometry()

        #-------------------- Preprocessing --------------------#
        S1_premerged = ee.ImageCollection("COPERNICUS/S1_GRD")
            .filterDate(start,end)
            .filterBounds(roi)
            .filter( ee.Filter.listContains('transmitterReceiverPolarisation',S1band))
            .sort('system:time_start')

        return S1_premerged
#%% (4) Function to exclude images that are not within 4 hours
def S1OptTempFilter(S1,Opt,hours):
        #-------------------- Temporal Filter --------------------#
        temporalFilter = ee.Filter.maxDifference(
                difference = hours * 60 * 60 * 1000,
                leftField = 'system:time_start',
                rightField = 'system:time_start')
        temporalJoin = ee.Join.saveAll(
                matchesKey = 'temporalMatches',
                measureKey = 'timeDiff',
                ordering = 'system:time_start',
                ascending = True)

        #--------------------    Matching    --------------------#
        temporalJoined = temporalJoin.apply(S1, Opt, temporalFilter)
```

```

        S1Matched = ee.ImageCollection(temporalJoined)

        return S1Matched
#%% (5) Export Metadata Per Coastline Segment
## (5.1) S1-Opt Matches
def S1OptMatchFunction(ROIs,S1band,start,end,cloudcover,whiteness,hours):
    def matchesPerSegment(ROIfeat,S1band,start,end,cloudcover,whiteness,hours):
        Opt = ee.ImageCollection(mergeOpticalData(ROIfeat,start,end,cloudcover,whiteness).get(2))
        S1processed = preProcessS1(ROIfeat,S1band,start,end)
        matchCol = S1OptTempFilter(S1processed,Opt,hours)
        return matchCol

    def saveMatchMetadata(ROIfeat,matchCol):
        S1Col = matchCol
        OptList = matchCol.aggregate_array('temporalMatches').flatten()
        return ee.Algorithms.If(
            condition = S1Col.size().gt(0),
            trueCase = ee.FeatureCollection(ee.List([
                S1Col.toList(S1Col.size()).map(lambda image: ee.Feature(ROIfeat.geometry(),{
                        'NAME': ee.String(ROIfeat.get('NAME')),
                        'SEG': ee.String(ROIfeat.get('SEG')),
                        'S1_UTC': ee.Number(ee.Image(image).get('system:time_start')),
                        'Opt_UTC':  ee.Number(-9999)})),
                OptList.map(lambda image: ee.Feature(ROIfeat.geometry(),{
                        'NAME': ee.String(ROIfeat.get('NAME')),
                        'SEG': ee.String(ROIfeat.get('SEG')),
                        'S1_UTC': ee.Number(-9999),
                        'Opt_UTC': ee.Number(ee.Image(image).get('system:time_start'))}))
                ]).flatten()),
            falseCase = ee.FeatureCollection([
                ee.Feature(ROIfeat.geometry(),{
                        'NAME': ee.String(ROIfeat.get('NAME')),
                        'SEG': ee.String(ROIfeat.get('SEG')),
                        'S1_UTC': ee.Number(-9999),
                        'Opt_UTC':  ee.Number(-9999)})
                ])
            )

    matchMetadata = ee.FeatureCollection(ROIs        .map(lambda feature: saveMatchMetadata(
            ee.Feature(feature),
            ee.ImageCollection(matchesPerSegment(ee.Feature(feature),S1band,start,end,cloudcover,whiteness,hours)))))\
        .flatten()

    return matchMetadata

S1OptMatchMetadata = S1OptMatchFunction(ROIs,S1band,start,end,cloudcover,whiteness,dt)

## Leave commented!!
## Export was necessary to create feature collections used in section below

# geemap.ee_export_vector_to_drive(
#     ee_object = S1OptMatchMetadata,
#     description = 'S1Opt_'+S1band+'_'+str(dt)+'hrs_Metadata_'+start[:4]+start[5:7]+start[8:10]+'_'+end[:4]+end[5:7]+end[8:10],
#     folder = 'Metadata',
#     file_format = 'shp')

## (5.2) All S1 Images
def countS1Function(ROIs,start,end):
    def saveS1Metadata(ROIfeat,S1):
        S1HH = S1.filter(ee.Filter.listContains('transmitterReceiverPolarisation','HH'))

        return ee.Algorithms.If(
            condition = S1HH.size().gt(0),
            trueCase = ee.FeatureCollection(ee.List([
                S1HH.toList(S1HH.size()).map(lambda image: ee.Feature(ROIfeat.geometry(),{
                        'NAME': ee.String(ROIfeat.get('NAME')),
                        'SEG': ee.String(ROIfeat.get('SEG')),
```

```
                        'S1_UTC': ee.Number(ee.Image(image).get('system:time_start')),
                        'Polarisation': ee.String(ee.List(ee.Image(image).get('
                            transmitterReceiverPolarisation')).get(-1)),
                        'Overpasses': S1HH.size()})),
                ]).flatten()),
            falseCase = ee.FeatureCollection([
                ee.Feature(ROIfeat.geometry(),{
                        'NAME': ee.String(ROIfeat.get('NAME')),
                        'SEG': ee.String(ROIfeat.get('SEG')),
                        'S1_UTC':  ee.Number(-9999),
                        'Polarisation': ee.String('NoData'),
                        'Overpasses': ee.Number(0)})
                ])
            )

    S1Metadata = ROIs
        .map(lambda feature: saveS1Metadata(
            ee.Feature(feature),
            preProcessS1(ee.Feature(feature),'HH',start,end)))\
        .flatten()

    return S1Metadata

S1Metadata = countS1Function(ROIs,start,end)

## Leave commented!!
## Export was necessary to create feature collections used in section below

# geemap.ee_export_vector_to_drive(
#     ee_object = S1Metadata,
#     description = 'S1_Metadata_'+start[:4]+start[5:7]+start[8:10]+'_'+end[:4]+end[5:7]+end
    [8:10],
#     folder = 'Metadata',
#     file_format = 'shp')


## (5.3) All Optical Images
def countOptFunction(ROIs,start,end,cloudcover,whiteness):
    def saveOptMetadata(ROIfeat,Opt):
        OptMetadata = ee.Algorithms.If(
            condition = Opt.size().gt(0),
            trueCase = ee.FeatureCollection(ee.List([
                Opt.toList(Opt.size()).map(lambda image: ee.Feature(ROIfeat.geometry(),{
                        'NAME': ee.String(ROIfeat.get('NAME')),
                        'SEG': ee.String(ROIfeat.get('SEG')),
                        'Opt_UTC': ee.Number(ee.Image(image).get('system:time_start')),
                        'Data': ee.String(ee.Image(image).get('Data')),
                        'Overpasses': Opt.size()})),
                ]).flatten()),
            falseCase = ee.FeatureCollection([
                ee.Feature(ROIfeat.geometry(),{
                        'NAME': ee.String(ROIfeat.get('NAME')),
                        'SEG': ee.String(ROIfeat.get('SEG')),
                        'Opt_UTC': ee.Number(-9999),
                        'Data': ee.String('NoData'),
                        'Overpasses': ee.Number(0)})
                ]))

        return OptMetadata

    OptMetadata = ROIs
        .map(lambda feature: saveOptMetadata(
            ee.Feature(feature),
            ee.ImageCollection(mergeOpticalData(ee.Feature(feature),start,end,cloudcover,
                whiteness).get(2))))\
        .flatten()

    return OptMetadata

OptMetadata = countOptFunction(ROIs,start,end,cloudcover,whiteness)
```

```
## Leave commented!!
## Export was necessary to create feature collections used in section below

# geemap.ee_export_vector_to_drive(
#     ee_object = OptMetadata,
#     description = ’Opt_Metadata_’+start[:4]+start[5:7]+start[8:10]+’_’+end[:4]+end[5:7]+end[8:10],
#     folder = ’Metadata’,
#     file_format = ’shp’)
```

B.3. Overpasses

3_Overpasses

```
#%% (1) Code set-up
## (1.1) Import packages
import ee
import geemap

## (1.2) Initialize
ee.Initialize()

## (1.3) Parameterization & Data Import
start = ’2014-01-01’
end = ’2022-04-01’
S1band = ’HV’
dt = 4

S1OptMetadata = ee.FeatureCollection(’users/skjeltmaps/S1Opt_’+S1band+’_’+str(dt)+’hrs_Metadata_’+start[:4]+start[5:7]+start[8:10]+’_’+end[:4]+end[5:7]+end[8:10])
OptMetadata = ee.FeatureCollection(’users/skjeltmaps/Opt_Metadata_’+start[:4]+start[5:7]+start[8:10]+’_’+end[:4]+end[5:7]+end[8:10])
S1Metadata = ee.FeatureCollection(’users/skjeltmaps/S1_Metadata_’+start[:4]+start[5:7]+start[8:10]+’_’+end[:4]+end[5:7]+end[8:10])

Opt_UTC = S1OptMetadata.aggregate_array(’Opt_UTC’)
S1_UTC = S1OptMetadata.aggregate_array(’S1_UTC’)

IceShelves = ee.FeatureCollection(”users/sophiederoda/IceShelf_Antarctica”)
ROIs = ee.FeatureCollection(’users/skjeltmaps/ISCLSegments20km’)

#%% (2) Reconstruct Match Data Set
## (2.1) General Functions

def addMetadata(image,utcname,metadata):
        utc = ee.String(image.get(’system:time_start’))
        filtered = metadata.filter(ee.Filter.inList(utcname,[utc]))
        name = filtered.aggregate_array(’NAME’).distinct()
        segment = name.map(lambda name: filtered.filterMetadata(’NAME’,’equals’,name).aggregate_array(’SEG’))
        wb = ee.List(ee.List(image.geometry(1e4).bounds(1e4).coordinates().get(0)).get(0)).get(0)
        return image.set(’NAME’,name).set(’SEG’,segment).set(’WB’,wb)

## (2.2) Reconstruction
S1 = ee.ImageCollection(”COPERNICUS/S1_GRD”)
    .filter(ee.Filter.inList(’system:time_start’,S1_UTC))
    .filter(ee.Filter.listContains(’transmitterReceiverPolarisation’,S1band))
    .map(lambda image: addMetadata(image,’S1_UTC’,S1OptMetadata))
    .sort(’system:time_start’)

L8 = ee.ImageCollection(”LANDSAT/LC08/C01/T2”)
    .filter(ee.Filter.inList(’system:time_start’,Opt_UTC))
    .map(lambda image: addMetadata(image,’Opt_UTC’,S1OptMetadata))
S2 = ee.ImageCollection(”COPERNICUS/S2”)
    .filter(ee.Filter.inList(’system:time_start’,Opt_UTC))
    .map(lambda image: addMetadata(image,’Opt_UTC’,S1OptMetadata))
Opt = L8.merge(S2).sort(’WB’)

## (2.3) Create S1-Opt Match Data Set
```

```python
def countTemporalMatches(S1Image,OptCol,hours):
        image = S1Image
        utc = image.get('system:time_start')
        names = ee.List(image.get('NAME'))
        segs = ee.List(image.get('SEG'))
        indeces = ee.List.sequence(0,names.size().subtract(1))
        filtered = ee.FeatureCollection(indeces.map(lambda i: S1OptMetadata.filterMetadata('NAME','equals',names.get(i)).filter(ee.Filter.inList('SEG',segs.get(i))))).flatten()
        coastlinegeometry = filtered.geometry()

        OptRoi = OptCol.filterBounds(coastlinegeometry)

        temporalFilter = ee.Filter.maxDifference(
                difference = hours * 60 * 60 * 1000,
                leftField = 'system:time_start',
                rightValue = utc)
        OptFiltered = OptRoi.filter(temporalFilter)

        return image.set('COUNT',OptFiltered.size())

S1OptDt = S1.map(lambda image: countTemporalMatches(image,Opt,dt))

#%% (3) Compute Overpasses
## (3.1) General Functions
def matchOverpasses(imCol,roi):
        segmentRoi = roi.geometry()
        segCol = imCol.filterBounds(segmentRoi)
        counts = segCol.aggregate_sum('COUNT')

        segmentOverpasses = ee.Feature(segmentRoi,{
            'NAME': ee.String(roi.get('NAME')),
            'SEG': ee.Number(roi.get('SEG')),
            'COUNTS': ee.Number(counts)})

        return segmentOverpasses

def collectionOverpasses(imCol,roi):
        segRoi = roi.geometry()
        segCol = imCol.filterBounds(segRoi)
        counts = segCol.size()

        segmentOverpasses = ee.Feature(segRoi,{
            'NAME': ee.String(roi.get('NAME')),
            'SEG': ee.Number(roi.get('SEG')),
            'COUNTS': ee.Number(counts)})

        return segmentOverpasses

def combineCollections(featCol1,featCol2,roi,ids):
    segRoi = roi.geometry()
    segCol1 = featCol1.filterBounds(segRoi)
    segCol2 = featCol2.filterBounds(segRoi)

    counts1 = ee.Number(segCol1.first().get('COUNTS'))
    counts2 = ee.Number(segCol2.first().get('COUNTS'))
    countsSum = counts1.add(counts2)

    combined = ee.Feature(segRoi,{
            'NAME': ee.String(roi.get('NAME')),
            'SEG': ee.Number(roi.get('SEG')),
            'COUNTS'+ids[0]: counts1,
            'COUNTS'+ids[1]: counts2,
            'COUNTSTOTAL': countsSum})
    return combined

## (3.2) Overpass Computation
S1OptOverpasses = ROIs.map(lambda roifeat: matchOverpasses(S1OptDt,roifeat))

L8Overpasses = ROIs.map(lambda roifeat: collectionOverpasses(OptMetadata.filterMetadata('Data','equals','L8'),roifeat))
```

```
S2Overpasses = ROIs.map(lambda roifeat: collectionOverpasses(OptMetadata.filterMetadata('Data','equals','S2'),roifeat))
OptOverpasses = ROIs.map(lambda roifeat: combineCollections(L8Overpasses,S2Overpasses,roifeat,['L8','S2']))

S1HHOverpasses = ROIs.map(lambda roifeat: collectionOverpasses(S1Metadata.filterMetadata('Polarisation','equals','HH'),roifeat))
S1HVOverpasses = ROIs.map(lambda roifeat: collectionOverpasses(S1Metadata.filterMetadata('Polarisation','equals','HV'),roifeat))
S1Overpasses = ROIs.map(lambda roifeat: combineCollections(S1HHOverpasses,S1HVOverpasses,roifeat,['HH','HV']))

#%% (4) Export Overpass Collections
## (4.1) S1-Opt Overpasses
## Leave commented!!
## Export was necessary to create feature collections used in section below

# geemap.ee_export_vector_to_drive(
#     ee_object = S1OptOverpasses,
#     description = 'S1Opt_'+S1band+'_'+str(dt)+'hrs_Overpasses_'+start[:4]+start[5:7]+start[8:10]+'_'+end[:4]+end[5:7]+end[8:10],
#     folder = 'OverpassesPerSegment',
#     file_format = 'shp')

## (4.2) Optical Overpasses
## Leave commented!!
## Export was necessary to create feature collections used in section below

# geemap.ee_export_vector_to_drive(
#     ee_object = OptOverpasses,
#     description = 'Opt_Overpasses_'+start[:4]+start[5:7]+start[8:10]+'_'+end[:4]+end[5:7]+end[8:10],
#     folder = 'OverpassesPerSegment',
#     file_format = 'shp')

## (4.3) S1 Overpasses
## Leave commented!!
## Export was necessary to create feature collections used in section below

# geemap.ee_export_vector_to_drive(
#     ee_object = S1Overpasses,
#     description = 'S1_Overpasses_'+start[:4]+start[5:7]+start[8:10]+'_'+end[:4]+end[5:7]+end[8:10],
#     folder = 'OverpassesPerSegment',
#     file_format = 'shp')
```

B.4. Match Data Set

4_ReferenceData

```
#%% (1) Code set-up
## (1.1) Import packages
import ee
import geemap
import numpy as np
import os

## (1.2) Initialize
ee.Initialize()

## (1.3) Parameterization & Data Import
start = '2014-01-01'
end = '2022-04-01'
crs = 'EPSG:3031'
S1band = 'HV'
dt = 4
dataset = 'S1Opt'

S1OptMetadata = ee.FeatureCollection('users/skjeltmaps/S1Opt_'+S1band+'_'+str(dt)+'hrs_Metadata_'+start[:4]+start[5:7]+start[8:10]+'_'+end[:4]+end[5:7]+end[8:10])
```

```
Metadata = eval(dataset+’Metadata’)

Opt_UTC = Metadata.aggregate_array(’Opt_UTC’)
S1_UTC = Metadata.aggregate_array(’S1_UTC’)

GroundingLine = ee.FeatureCollection(”users/sophiederoda/GroundingLine_Antarctica”)
DEM = ee.Image(’CPOM/CryoSat2/ANTARCTICA_DEM’).select(’elevation’)
IceShelves = ee.FeatureCollection(”users/sophiederoda/IceShelf_Antarctica”)
ROIs = ee.FeatureCollection(’users/skjeltmaps/ISCLSegments20km’)

AmundsenEmbayment = ee.Geometry.Polygon(
    coords = [[[-108.406159, -75.468542], [-108.406159, -71.238725], [-96.219667, -71.238725], [-96.219667, -75.468542], [-108.406159, -75.468542]]],
    geodesic = False,
    proj = ’EPSG:4326’)
CarneyCoast = ee.Geometry.Polygon(
    coords = [[[-146.815157, -74.904569], [-142.211175, -76.343611], [-128.293796, -74.877073], [-120.513741, -75.203774], [-120.513741, -73.617891], [-127.520609, -72.722358], [-146.815157, -74.904569]]],
    geodesic = False,
    proj = ’EPSG:4326’)

trainingName = [’AmundsenEmbayment’]
trainingArea = AmundsenEmbayment

validationName = [’CarneyCoast’]
validationArea = CarneyCoast

trainValArea = trainingArea.union(validationArea,maxError=1e3)

L8name = ’L8’
L8bands = [’B4’,’B3’,’B2’]
L8_params = {
  ’bands’: L8bands,
  ’min’: 0,
  ’max’: 30000}

S2name = ’S2’
S2bands = [’B4’, ’B3’, ’B2’]
S2_params = {
  ’min’: 0,
  ’max’: 0.3*1e4,
  ’bands’: S2bands}

RGBbands = [’Red’,’Green’,’Blue’]
RGB_params = {
  ’bands’: RGBbands,
  ’min’: 0,
  ’max’: 3e4}

S1name = ’S1’
HHmin = -30
HHmax = 2.5
HVmin = -40
HVmax = -3

S1_HH_params = {
    ’min’: HHmin,
    ’max’: HHmax}
S1_HV_params = {
    ’min’: HVmin,
    ’max’: HVmax}
S1_RGB_params = {
    ’min’: [HVmin,HHmin,HHmin],
    ’max’: [HVmax,HHmax,HHmax]}
angle_params = {
    ’min’: 0,
    ’max’: 90}

## Edit visualization export folder is necessary
# html_dir = os.path.join(os.path.expanduser(’~’),’Downloads’)
```

```
# if not os.path.exists(html_dir):
#     os.makedirs(html_dir)

#%% (2) Reconstruct Match Data Set
## (2.1) Cloud Mask Functions
def cloudMaskL8(image):
    def getQABits(image, bitStart, bitEnd, name):
        # Compute the bits we need to extract.
        pattern = 0
        for i in range(bitStart,bitEnd+1):
            pattern += 2**i
        return ee.Image(image).select([0], [name]).bitwiseAnd(pattern).rightShift(bitStart)

    QA = ee.Image(image).select('BQA').toInt()
    cloud = getQABits(QA,4,4,'CLOUD')
    cloud_confidence = getQABits(QA,5,6,'CLOUD_CONFIDENCE')
    cloudshadow_confidence = getQABits(QA,7,8,'CLOUDSHADOW_CONFIDENCE')
    cirrus_confidence = getQABits(QA,11,12,'CIRRUS_CONFIDENCE')

    return ee.Image(image).updateMask(cloud.eq(0))
    .updateMask(cloud_confidence.lt(3))
    .updateMask(cloudshadow_confidence.lt(3))
    .updateMask(cirrus_confidence.lt(3))

def cloudMaskS2(image):
    qa = ee.Image(image).select('QA60').toInt()

    cloudBitMask = 1 << 10
    cirrusBitMask = 1 << 11

    cloud = qa.bitwiseAnd(cloudBitMask).eq(0)
    cirrus = qa.bitwiseAnd(cirrusBitMask).eq(0)

    return ee.Image(image)
        .updateMask(cloud)
        .updateMask(cirrus)

## (2.2) General Functions
def utcToLocal(image):
    centroid = ee.Image(image).geometry(maxError=1e3).centroid(maxError=1e3)
    longitude = centroid.coordinates().get(0)
    localTime = ee.Image(image).date().advance(ee.Number(longitude).divide(15).ceil().
        subtract(1),'hour')
    localMillis = ee.Date(localTime).millis()

    return image
        .set('utctime',ee.Image(image).get('system:time_start'))
        .set('localtime',localMillis)

def addID(image,name):
    id = ee.String(name+'_').cat(ee.String(ee.Date(ee.Image(image).get('localtime')).format('
        YYYY_MM_dd_KK:mma')))
    return image.set('id',id).set('data',name)

def reproject(image):
    scale = image.select(0).projection().nominalScale()
    return image.reproject(crs=crs,scale=scale)


def addRGBbands(image,bandnames):
    renamed = image.select(bandnames,RGBbands)
    rgb = image.addBands(renamed).select(RGBbands)
    return rgb

def addMetadata(image,utcname,metadata):
        utc = ee.String(image.get('system:time_start'))
        filtered = metadata.filter(ee.Filter.inList(utcname,[utc]))
        name = filtered.aggregate_array('NAME').distinct()
        segment = name.map(lambda name: filtered.filterMetadata('NAME','equals',name).
            aggregate_array('SEG'))
        wb = ee.List(ee.List(image.geometry(1e4).bounds(1e4).coordinates().get(0)).get(0)).
```

```
            get(0)
        return image.set('NAME',name).set('SEG',segment).set('WB',wb)

def angleNormalization(image):
    def degreesToRadians(object):
        return object.multiply(2*np.pi).divide(180)

    thetaRef = degreesToRadians(ee.Number(30))
    radians = degreesToRadians(image.select('angle'))

    HH = image.select('HH').multiply(thetaRef.cos().pow(ee.Number(2))).divide(radians.cos().pow(ee.Number(2)))
    HV = image.select('HV').multiply(thetaRef.cos().pow(ee.Number(2))).divide(radians.cos().pow(ee.Number(2)))

    normalized = image.addBands(HH,overwrite=True).addBands(HV,overwrite=True)

    return normalized

## (2.3) Reconstruction
S1 = ee.ImageCollection("COPERNICUS/S1_GRD")
    .filter(ee.Filter.inList('system:time_start',S1_UTC))
    .filter(ee.Filter.listContains('transmitterReceiverPolarisation',S1band))
    .map(lambda image: addMetadata(image,'S1_UTC',Metadata))
    .map(utcToLocal)
    .map(lambda image: addID(image,S1name))
    .sort('WB')
    .filterBounds(trainValArea)

L8 = ee.ImageCollection("LANDSAT/LC08/C01/T2")
    .filter(ee.Filter.inList('system:time_start',Opt_UTC))
    .map(lambda image: addRGBbands(image,L8bands))
    .map(lambda image: addMetadata(image,'Opt_UTC',Metadata))
    .map(utcToLocal)
    .map(lambda image: addID(image,L8name))
    .filterBounds(trainValArea)
S2 = ee.ImageCollection("COPERNICUS/S2")
    .filter(ee.Filter.inList('system:time_start',Opt_UTC))
    .map(lambda image: addRGBbands(image,S2bands))
    .map(lambda image: addMetadata(image,'Opt_UTC',Metadata))
    .map(utcToLocal)
    .map(lambda image: addID(image,S2name))
    .filterBounds(trainValArea)
Opt = L8.merge(S2).sort('WB')
OptList = Opt.toList(Opt.size())

## (2.4) Quality Assessment of Optical Images
## Leave commented!!
## Visualization was necessary to assess image quality

# Map = geemap.Map()
# Map.centerObject(Opt.first().geometry(),7)
# for i in range(OptList.size().getInfo()):
#     image = ee.Image(OptList.get(i))
#     Map.addLayer(image,RGB_params,'Opt'+str(i+1))
#     print('check '+str(i+1))

# html_file = os.path.join(html_dir+'\QualityAssessment\Training_Validation\Optical_'+S1band+'_'+str(dt)+'hrs_'+start[:4]+start[5:7]+start[8:10]+'_'+end[:4]+end[5:7]+end[8:10]+'.html')
# Map.to_html(outfile=html_file, title= 'Opt images', width='100%', height='880px')

indexList = ee.List.sequence(0,OptList.size().subtract(1))
poorQualityIndeces_HV_4hrs = ee.List([5,14,17,23,24,28,29,30])
goodQualityIndeces_HV_4hrs = indexList.removeAll(poorQualityIndeces_HV_4hrs)

qOptList_HV_4hrs = goodQualityIndeces_HV_4hrs.map(lambda index: OptList.get(index))
qOpt_HV_4hrs = ee.ImageCollection(qOptList_HV_4hrs)

qOpt = qOpt_HV_4hrs
qOptList = qOpt.toList(qOpt.size())
```

```

## Leave commented!!
## Visualization was necessary to assess image quality

# Map = geemap.Map()
# Map.centerObject(qOpt.first().geometry(),7)
# for i in range(qOptList.size().getInfo()):
#     image = ee.Image(qOptList.get(i))
#     Map.addLayer(image,RGB_params,'Opt'+str(i+1))
#     print('check '+str(i+1))

# html_file = os.path.join(html_dir+'\QualityAssessment\Training_Validation\OpticalQuality_'+
    S1band+'_'+str(dt)+'hrs_'+start[:4]+start[5:7]+start[8:10]+'_'+end[:4]+end[5:7]+end
    [8:10]+'.html')
# Map.to_html(outfile=html_file, title= 'Opt images', width='100%', height='880px')


#%% (3) Visualization of Match Data Set
## (3.1) Functions
def findMatches(image,imCol,hours):
        utc = image.get('system:time_start')
        temporalFilter = ee.Filter.maxDifference(
                difference = hours * 60 * 60 * 1000,
                leftField = 'system:time_start',
                rightValue = utc)

        names = ee.List(image.get('NAME'))
        segs = ee.List(image.get('SEG'))
        indeces = ee.List.sequence(0,names.size().subtract(1))
        filtered = ee.FeatureCollection(indeces.map(lambda i: ROIs.filterMetadata('NAME','
            equals',names.get(i)).filter(ee.Filter.inList('SEG',segs.get(i))))).flatten()
        geometry = filtered.geometry()
        filteredCol = imCol.filterBounds(geometry).filter(temporalFilter)

        return image.set('matches',filteredCol.size()).set('utcs',filteredCol.aggregate_array
            ('system:time_start')).set('iscl',geometry)

def addMatchesAsBands(image,imCol,data):
        utcs = ee.List(image.get('utcs'))
        matches =  imCol.filter(ee.Filter.inList('system:time_start',utcs))
        matchList = matches.toList(matches.size())

        def addMatches(match,iterations):
                iteration = ee.List(iterations)
                image = ee.Image(iteration.get(0))
                namelist = ee.List(iteration.get(1))

                matched = ee.Image(match)
                id = ee.String(matched.get('id'))
                utc = ee.String(matched.get('system:time_start'))
                fp = matched.geometry()

                index = matchList.indexOf(matched)
                name = ee.String(namelist.get(index))
                idname = ee.String(data).cat(ee.String('id')).cat(name)
                utcname = ee.String(data).cat(ee.String('utc')).cat(name)
                fpname = ee.String(data).cat(ee.String('fp')).cat(name)

                bandnames = matched.bandNames()
                updatednames = bandnames.map(lambda bandname: ee.String(bandname).cat(name))

                bandsAdded = image
                    .addBands(matched.select(bandnames,updatednames))
                    .set(idname,id)
                    .set(utcname,utc)
                    .set(fpname,fp)

                return ee.List([bandsAdded,namelist])

        S1nameList = ee.List(['_1','_2','_3','_4','_5','_6','_7','_8','_9','_10'])
        iterList = ee.List([image,S1nameList])
```

```
        matchesAdded = ee.Image(ee.List(matchList.iterate(
                function = addMatches,
                first = iterList
        )).get(0))

        return matchesAdded

def filterAllMatches(imList,imCol,hours):
        dataToMatch = ee.String(imCol.aggregate_array('data').distinct().get(0))
        matchesCol = ee.ImageCollection(imList)
            .map(lambda image: findMatches(image,imCol,hours))
            .filterMetadata('matches','greater_than',0)
            .map(lambda image: addMatchesAsBands(image,imCol,dataToMatch))

        return matchesCol

def mapS1Opt(S1Opt,S1band,dt,region):
    geometry = S1Opt.first().geometry()

    Map = geemap.Map()
    Map.centerObject(geometry,6)

    S1OptList = S1Opt.toList(S1Opt.size())

    for i in range(S1OptList.size().getInfo()):
            image = ee.Image(S1OptList.get(i))
            utc = ee.Date(image.get('system:time_start'))
            id = ee.String(image.get('id')).getInfo()
            bands = image.bandNames()
            numberOfMatches = bands.filter(ee.Filter.stringContains('item','Red')).size().
                getInfo()

            HHname = str(i+1)+'_S1_HH | Time (local): '+id[-18:]
            Map.addLayer(image.select('HH'),S1_HH_params,HHname)
            print('S1 HH image '+str(i+1)+' added')

            for k in range(numberOfMatches):
                utcname = image.propertyNames().filter(ee.Filter.stringEndsWith('item','utc_'
                    +str(k+1))).get(0).getInfo()
                Optutc = ee.Date(image.get(utcname))
                timeDiff = int(np.ceil(utc.difference(Optutc,'minute').getInfo()))
                name = str(i+1)+'_'+str(k+1)+'_'+utcname[:2]+' | '+'\u0394'+'t = '+str(
                    timeDiff)+'min'
                Map.addLayer(image.select(bands.filter(ee.Filter.stringEndsWith('item',str(k
                    +1)))),RGBbands),RGB_params,name)
                print(utcname[:2]+' image '+str(k+1)+' added')

            names = ee.List(image.get('NAME'))
            segs = ee.List(image.get('SEG'))
            indeces = ee.List.sequence(0,names.size().subtract(1))
            filtered = ee.FeatureCollection(indeces
                .map(lambda i: ROIs
                    .filterBounds(trainValArea)
                    .filterMetadata('NAME','equals',names.get(i))
                    .filter(ee.Filter.inList('SEG',segs.get(i)))))
                .flatten()
            coastlinegeometry = filtered.geometry()

            Map.addLayer(coastlinegeometry,{},str(i+1)+'_IceShelfCoastLine',opacity=0.25)

    Map.addLayer(GroundingLine.filterBounds(trainValArea).geometry(),{},'Grounding Line',
        opacity=0.15)
    Map.addLayer(IceShelves.filterBounds(trainValArea).geometry(),{},'Ice Shelves',opacity
        =0.15)

    html_file = os.path.join(html_dir+'\QualityAssessment\Training_Validation\S1OptMatches_'+
        region+'_'+S1band+'_'+str(dt)+'hrs_'+start[:4]+start[5:7]+start[8:10]+'_'+end[:4]+end
        [5:7]+end[8:10]+'.html')
    Map.to_html(outfile=html_file, title= 'S1 & Opt Matches', width='100%', height='880px')
    print('Map finished')

```

```
    return Map

## (3.2) Create Reference Data Set
S1OptDt = filterAllMatches(S1,qOpt,dt).sort('WB')
S1OptList = S1OptDt.toList(S1OptDt.size())

TrainingData = filterAllMatches(S1,qOpt.filterBounds(trainingArea),dt)
ValidationData = filterAllMatches(S1,qOpt.filterBounds(validationArea),dt)

## (3.3) Visualize & Select Quality Matches
## Leave commented!!
## Used to visualize data set

# QualityMap = mapS1Opt(S1OptDt,S1band,dt,'Training&Validation')

## Poor quality indeces: [0,3,10,14,15,18,25]

## (3.4) Manual Selection of Quality Matches
indexList = ee.List.sequence(0,S1OptList.size().subtract(1))
poorQualityIndeces_HV_Opt_4hrs = ee.List([0,3,10,14,15,18,25])
goodQualityIndeces_HV_Opt_4hrs = indexList.removeAll(poorQualityIndeces_HV_Opt_4hrs)

qS1OptList_HV_4hrs = goodQualityIndeces_HV_Opt_4hrs.map(lambda index: ee.Image(S1OptList.get(
    index)))
qS1Opt_HV_4hrs = ee.ImageCollection(qS1OptList_HV_4hrs)

matchIndex = ee.List.sequence(0,qS1Opt_HV_4hrs.size().subtract(1))
S1_utc = qS1Opt_HV_4hrs.aggregate_array('system:time_start').flatten().distinct()
Opt_utcs = qS1Opt_HV_4hrs.aggregate_array('utcs')

S1Quality = ee.ImageCollection(
    matchIndex.map(lambda i: S1\
        .filter(ee.Filter.inList('system:time_start',ee.List([S1_utc.get(i)])))\
        .toList(ee.List([S1_utc.get(i)]).size())\
        .map(lambda image: ee.Image(image).set('match',ee.Number(i).add(1)))
    ).flatten()
)
OptQuality = ee.ImageCollection(
    matchIndex.map(lambda i: Opt\
        .filter(ee.Filter.inList('system:time_start',ee.List(Opt_utcs.get(i))))\
        .toList(ee.List(Opt_utcs.get(i)).size())\
        .map(lambda image: ee.Image(image).set('match',ee.Number(i).add(1)))
    ).flatten()
)

## Leave commented!!
## Used to visualize data set

# RefMap = mapS1Opt(qS1Opt_HV_4hrs,S1band,dt,'Reference_Data')

#%% (4) Store Training & Validation Data Set
S1_FC = ee.FeatureCollection(S1Quality.toList(S1Quality.size()).map(lambda image:
    ee.Feature(ee.Image(image).geometry(),{
        'utc': ee.Image(image).get('system:time_start'),
        'data': ee.Image(image).get('data'),
        'match': ee.Image(image).get('match')})
))
Opt_FC = ee.FeatureCollection(OptQuality.toList(OptQuality.size()).map(lambda image:
    ee.Feature(ee.Image(image).geometry(),{
        'utc': ee.Image(image).get('system:time_start'),
        'data': ee.Image(image).get('data'),
        'match': ee.Image(image).get('match')})
))

referenceData = S1_FC.merge(Opt_FC)
## Leave commented!!
## Export was necessary to create feature collections used in section below

# geemap.ee_export_vector_to_drive(
#     ee_object = referenceData,
#     description = 'RD_'+S1band+'_'+str(dt)+'hrs_'+start[:4]+start[5:7]+start[8:10]+'_'+end
```

```
    [:4]+end[5:7]+end[8:10],
#     folder = 'TrainingValidation',
#     file_format = 'shp')
```

B.5. Training & Validation Data

5_TrainingValidationData

```
#%% (1) Code set-up
## (1.1) Import packages
import ee
import geemap
import numpy as np
import os

## (1.2) Initialize
ee.Initialize()

## (1.3) Parameterization & Data Import
start = '2014-01-01'
end = '2022-04-01'
crs = 'EPSG:3031'
S1band = 'HV'
dt = 4
elevation_threshold = 200

ReferenceData = ee.FeatureCollection('users/skjeltmaps/RD_'+S1band+'_'+str(dt)+'hrs_'+start
    [:4]+start[5:7]+start[8:10]+'_'+end[:4]+end[5:7]+end[8:10])
S1OptMetadata = ee.FeatureCollection('users/skjeltmaps/S1Opt_'+S1band+'_'+str(dt)+'
    hrs_Metadata_'+start[:4]+start[5:7]+start[8:10]+'_'+end[:4]+end[5:7]+end[8:10])

S1_UTC = ReferenceData.filterMetadata('data','equals','S1').aggregate_array('utc')
Opt_UTC = ReferenceData.filterMetadata('data','not_equals','S1').aggregate_array('utc')

DEM = ee.Image('CPOM/CryoSat2/ANTARCTICA_DEM').select('elevation')
IceShelves = ee.FeatureCollection("users/sophiederoda/IceShelf_Antarctica")
ROIs = ee.FeatureCollection('users/skjeltmaps/ISCLSegments20km')

AmundsenEmbayment = ee.Geometry.Polygon(
    coords = [[[-108.406159, -75.468542], [-108.406159, -71.238725], [-96.219667,
        -71.238725], [-96.219667, -75.468542], [-108.406159, -75.468542]]],
    geodesic = False,
    proj = 'EPSG:4326')
CarneyCoast = ee.Geometry.Polygon(
    coords = [[[-146.815157, -74.904569], [-142.211175, -76.343611], [-128.293796,
        -74.877073], [-120.513741, -75.203774], [-120.513741, -73.617891], [-127.520609,
        -72.722358], [-146.815157, -74.904569]]],
    geodesic = False,
    proj = 'EPSG:4326')

trainingName = ['AmundsenEmbayment']
trainingArea = AmundsenEmbayment

validationName = ['CarneyCoast']
validationArea = CarneyCoast

trainValArea = trainingArea.union(validationArea,maxError=1e3)

L8name = 'L8'
L8bands = ['B4','B3','B2']
L8_params = {
  'bands': L8bands,
  'min': 0,
  'max': 30000}

S2name = 'S2'
S2bands = ['B4', 'B3', 'B2']
S2_params = {
  'min': 0,
  'max': 0.3*1e4,
```

```
  'bands': S2bands}

RGBbands = ['Red','Green','Blue']
RGB_params = {
  'bands': RGBbands,
  'min': 0,
  'max': 3e4}

S1name = 'S1'
HHmin = -35
HHmax = 5
HVmin = -40
HVmax = 0

S1_HH_params = {
'min': HHmin,
'max': HHmax}
S1_HV_params = {
'min': HVmin,
'max': HVmax}
S1_RGB_params = {
'min': [HVmin,HHmin,HHmin],
'max': [HVmax,HHmax,HHmax]}
angle_params = {
    'min': 0,
    'max': 90}

## Edit visualization export folder is necessary
# html_dir = os.path.join(os.path.expanduser('~'),'Downloads')
# if not os.path.exists(html_dir):
#     os.makedirs(html_dir)

#%% (2) Reconstruct Match Data Set
## (2.1) General Functions
def utcToLocal(image):
    centroid = ee.Image(image).geometry(maxError=1e3).centroid(maxError=1e3)
    longitude = centroid.coordinates().get(0)
    localTime = ee.Image(image).date().advance(ee.Number(longitude).divide(15).ceil().
        subtract(1),'hour')
    localMillis = ee.Date(localTime).millis()

    return image
        .set('utctime',ee.Image(image).get('system:time_start'))
        .set('localtime',localMillis)

def addID(image,name):
    id = ee.String(name+'_').cat(ee.String(ee.Date(ee.Image(image).get('localtime')).format('
        YYYY_MM_dd_KK:mma')))
    return image.set('id',id).set('data',name)

def addRGBbands(image,bandnames):
    renamed = image.select(bandnames,RGBbands)
    rgb = image.addBands(renamed).select(RGBbands)
    return rgb

def addWB(image):
        wb = ee.List(ee.List(image.geometry(1e4).bounds(1e4).coordinates().get(0)).get(0)).
            get(0)
        return image.set('WB',wb)

def addMetadata(image,utcname,metadata):
        utc = ee.String(image.get('system:time_start'))
        filtered = metadata.filter(ee.Filter.inList(utcname,[utc]))
        name = filtered.aggregate_array('NAME').distinct()
        segment = name.map(lambda name: filtered.filterMetadata('NAME','equals',name).
            aggregate_array('SEG'))
        wb = ee.List(ee.List(image.geometry(1e4).bounds(1e4).coordinates().get(0)).get(0)).
            get(0)
        return image.set('NAME',name).set('SEG',segment).set('WB',wb)

def addMatchIndex(image,metadata):
```

```
    utc = image.get('system:time_start')
    match = metadata.filterMetadata('utc','equals',utc).first()
    matchIndex = ee.Number(match.get('match'))
    return image.set('match',matchIndex)

def angleNormalization(image):
    def degreesToRadians(object):
        return object.multiply(np.pi).divide(180)

    thetaRef = degreesToRadians(ee.Number(30))
    radians = degreesToRadians(image.select('angle'))

    angleCorr = image.select('angle').divide(thetaRef.cos().pow(ee.Number(2))).multiply(
        radians.cos().pow(ee.Number(2)))
    HH = image.select('HH').divide(thetaRef.cos().pow(ee.Number(2))).multiply(radians.cos().
        pow(ee.Number(2)))
    HV = image.select('HV').divide(thetaRef.cos().pow(ee.Number(2))).multiply(radians.cos().
        pow(ee.Number(2)))

    normalized = image.addBands(HH,overwrite=True).addBands(HV,overwrite=True).addBands(
        angleCorr)

    return normalized

## (2.2) Reconstruction
S1 = ee.ImageCollection("COPERNICUS/S1_GRD")
    .filter(ee.Filter.inList('system:time_start',S1_UTC))
    .filter(ee.Filter.listContains('transmitterReceiverPolarisation',S1band))
    .map(utcToLocal)
    .map(angleNormalization)
    .map(lambda image: image.updateMask(DEM.unmask().gt(elevation_threshold).Not()))
    .map(lambda image: addID(image,S1name))
    .map(lambda image: addMetadata(image,'S1_UTC',S1OptMetadata))
    .map(lambda image: addMatchIndex(image,ReferenceData))
    .sort('WB')

L8 = ee.ImageCollection("LANDSAT/LC08/C01/T2")
    .filter(ee.Filter.inList('system:time_start',Opt_UTC))
    .map(lambda image: addRGBbands(image,L8bands))
    .map(utcToLocal)
    .map(lambda image: addID(image,L8name))
    .map(lambda image: addMatchIndex(image,ReferenceData))
    .map(lambda image: addMetadata(image,'Opt_UTC',S1OptMetadata))
S2 = ee.ImageCollection("COPERNICUS/S2")
    .filter(ee.Filter.inList('system:time_start',Opt_UTC))
    .map(lambda image: addRGBbands(image,S2bands))
    .map(utcToLocal)
    .map(lambda image: addID(image,S2name))
    .map(lambda image: addMatchIndex(image,ReferenceData))
    .map(lambda image: addMetadata(image,'Opt_UTC',S1OptMetadata))
Opt = L8.merge(S2).sort('WB')
OptList = Opt.toList(Opt.size())

#%% (3) Create Training & Validation Data Set
## (3.1) Match functions
def matchDataSets(S1,Opt):
    def addMatchesAsBands(imgS1,Opt):
        matchIndex = ee.Number(imgS1.get('match'))
        OptCol = Opt.filterMetadata('match','equals',matchIndex)
        OptList = OptCol.toList(OptCol.size())
        GeomList = OptList.map(lambda image: ee.Image(image).geometry())

        def combineGeometries(geom2,geom1):
            geom = ee.Geometry(geom1).union(ee.Geometry(geom2),maxError=1e3)
            return geom

        OptGeometry = ee.Geometry(GeomList.iterate(
            function = combineGeometries,
            first = ee.Geometry(GeomList.get(0))
        ))
        S1Geometry = imgS1.geometry()
```

```
        OptImage = OptCol.median().clip(S1Geometry)
        S1Image = imgS1.clip(OptGeometry)
        matchAdded = S1Image.addBands(OptImage)

        return matchAdded

    matchesOfInterest = Opt.aggregate_array('match').distinct().sort()
    S1Filtered = S1.filter(ee.Filter.inList('match',matchesOfInterest))
    S1Opt = S1Filtered.map(lambda imgS1: addMatchesAsBands(imgS1,Opt))

    return S1Opt

## (3.2) Create Reference Data Set
Opt_Amundsen = Opt.filterBounds(AmundsenEmbayment)
Opt_Carney = Opt.filterBounds(CarneyCoast)

AmundsenMatches = Opt_Amundsen.aggregate_array('match').distinct().sort()
CarneyMatches = Opt_Carney.aggregate_array('match').distinct().sort()

S1_Amundsen = S1.filter(ee.Filter.inList('match',AmundsenMatches))
Opt_Amundsen = Opt.filter(ee.Filter.inList('match',AmundsenMatches))
S1Opt_Amundsen = matchDataSets(S1_Amundsen,Opt_Amundsen)

S1_Carney = S1.filter(ee.Filter.inList('match',CarneyMatches))
Opt_Carney = Opt.filter(ee.Filter.inList('match',CarneyMatches))
S1Opt_Carney = matchDataSets(S1_Carney,Opt_Carney)

trainIDsAmundsen = ee.List([1,2,5,6,7,8])
trainIDsCarney = ee.List([1,4,11])

S1Training = ee.ImageCollection(trainIDsAmundsen.map(lambda index: ee.Image(S1_Amundsen.
    toList(S1_Amundsen.size()).get(index))))
    .merge(ee.ImageCollection(trainIDsCarney.map(lambda index: ee.Image(S1_Carney.toList(
        S1_Carney.size()).get(index)))))
OptTraining = Opt.filter(ee.Filter.inList('match',S1Training.aggregate_array('match').
    distinct().sort()))
TrainingData = ee.ImageCollection(trainIDsAmundsen        .map(lambda index: ee.Image(
    S1Opt_Amundsen.toList(S1Opt_Amundsen.size()).get(index))))
    .merge(ee.ImageCollection(trainIDsCarney.map(lambda index: ee.Image(S1Opt_Carney.toList(
        S1Opt_Carney.size()).get(index)))))

valIDsAmundsen = ee.List([0,3,4])
valIDsCarney = ee.List([10])

S1Validation = ee.ImageCollection(valIDsAmundsen.map(lambda index: ee.Image(S1_Amundsen.
    toList(S1_Amundsen.size()).get(index))))
    .merge(ee.ImageCollection(valIDsCarney.map(lambda index: ee.Image(S1_Carney.toList(
        S1_Carney.size()).get(index)))))
OptValidation = Opt.filter(ee.Filter.inList('match',S1Validation.aggregate_array('match').
    distinct().sort()))
ValidationData = ee.ImageCollection(valIDsAmundsen        .map(lambda index: ee.Image(
    S1Opt_Amundsen.toList(S1Opt_Amundsen.size()).get(index))))
    .merge(ee.ImageCollection(valIDsCarney.map(lambda index: ee.Image(S1Opt_Carney.toList(
        S1Opt_Carney.size()).get(index)))))

#%% (3) Creation of Sampling Polygons
## (3.1) Training Polygons
OW1aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-114.226429, -73.138338], [-113.743186, -73.071926], [-113.435668,
        -73.217192], [-113.374165, -73.371302], [-113.330234, -73.543974], [-113.980414,
        -73.308308], [-114.226429, -73.138338]]
)
OW1bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-105.2167, -74.145339], [-105.01242, -74.206453], [-105.315545, -74.179519],
        [-105.434159, -74.162138], [-105.649421, -74.163337], [-105.447338, -74.150141],
        [-105.276007, -74.154341], [-105.330921, -74.138134], [-105.223289, -74.13453],
```

```
        [-105.2167, -74.145339]]
)
OW1cT = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-110.213527, -72.875053], [-110.086127, -72.889282], [-110.125665,
        -72.917704], [-110.222314, -72.940925], [-110.323355, -72.957677], [-110.406824,
        -72.969263], [-110.42879, -73.000128], [-110.367286, -73.025805], [-110.437576,
        -73.041193], [-110.49908, -73.005265], [-110.406824, -72.947369], [-110.296997,
        -72.933187], [-110.213527, -72.875053]]
)
OW1T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = ee.List([OW1aT,OW1bT,OW1cT])
)
DW1aT = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-106.927888, -74.224536], [-107.059681, -74.291298], [-106.796094,
        -74.369635], [-105.50452, -74.5017], [-105.346367, -74.452303], [-104.915842,
        -74.454661], [-105.003704, -74.279395], [-105.645099, -74.226921], [-105.8999,
        -74.262718], [-106.664301, -74.272254], [-106.822453, -74.229313], [-106.927888,
        -74.224536]]
)
DW1bT = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-113.177057, -73.116974], [-112.948616, -73.34015], [-113.05405, -73.683134],
        [-112.53127, -73.927035], [-111.33195, -73.973205], [-111.441778, -73.836756],
        [-112.149069, -73.660895], [-112.280863, -73.429366], [-112.026062, -73.328806],
        [-111.551606, -73.235262], [-111.850337, -73.118251], [-113.177057, -73.116974]]
)
DW1cT = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-110.442138, -73.215918], [-110.398207, -73.200685], [-110.422369, -73.17908],
         [-110.464103, -73.191156], [-110.514624, -73.189249], [-110.551966, -73.194969],
        [-110.53, -73.211477], [-110.442138, -73.215918]]
)
DW1T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = ee.List([DW1aT,DW1bT,DW1cT])
)
FI1aT = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-110.395281, -73.919426], [-110.118515, -73.916992], [-110.131695,
        -73.882869], [-109.920825, -73.809511], [-109.876894, -73.83767], [-109.749494,
        -73.803383], [-109.841749, -73.773939], [-109.775853, -73.748131], [-109.872501,
        -73.693931], [-109.96915, -73.70873], [-110.030653, -73.687761], [-110.008688,
        -73.628409], [-109.947184, -73.588725], [-109.850536, -73.592447], [-109.868108,
        -73.566361], [-109.929612, -73.571332], [-109.837356, -73.514065], [-109.846142,
        -73.492849], [-109.95597, -73.512817], [-109.894467, -73.541478], [-109.96915,
        -73.541478], [-110.021867, -73.582516], [-110.004295, -73.609818], [-110.118515,
        -73.609818], [-110.329385, -73.625932], [-110.311812, -73.655637], [-110.390888,
        -73.665528], [-110.346957, -73.702565], [-110.197591, -73.654399], [-110.158054,
        -73.675413], [-110.250309, -73.691461], [-110.092157, -73.778849], [-110.390888,
        -73.810737], [-110.29424, -73.840116], [-110.201985, -73.833999], [-110.197591,
        -73.864559], [-110.329385, -73.873106], [-110.412854, -73.895063], [-110.395281,
        -73.919426]]
)
FI1bT = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-107.364818, -73.575374], [-107.795343, -73.66831], [-107.518577, -73.695475],
         [-107.584473, -73.781614], [-107.298921, -73.769335], [-107.11441, -73.829412],
        [-107.022155, -73.785295], [-107.136376, -73.755817], [-107.017762, -73.729983],
        [-106.66192, -73.84287], [-106.437871, -73.828188], [-106.560878, -73.784068],
        [-106.679492, -73.800014], [-106.692672, -73.785295], [-106.569665, -73.74352],
```

```
        [-106.727816, -73.679428], [-106.762961, -73.529336], [-107.022155, -73.486921],
        [-107.364818, -73.575374]]
)
FI1cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-105.619598, -73.844705], [-105.364798, -73.930076], [-105.404336,
        -73.973812], [-105.38237, -74.000481], [-105.72064, -73.992], [-105.707461,
        -74.0742], [-106.137986, -74.030734], [-106.076482, -73.998058], [-106.124806,
        -73.922775], [-106.274172, -73.900857], [-106.287352, -73.847149], [-106.11602,
        -73.839814], [-105.79093, -73.810432], [-105.619598, -73.844705]]
)
FI1dT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-106.9925, -74.485839], [-107.194583, -74.521063], [-107.273659, -74.602949],
        [-107.38788, -74.692535], [-106.917817, -74.658853], [-106.983713, -74.609946],
        [-106.737699, -74.598279], [-106.540009, -74.657692], [-106.966141, -74.749278],
        [-106.838741, -74.812726], [-106.192954, -74.828837], [-106.197347, -74.635583],
        [-106.403823, -74.592443], [-106.395037, -74.542157], [-106.9925, -74.485839]]
)
FI1eT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-105.164965, -74.595655], [-104.940916, -74.753615], [-104.391777,
        -74.858425], [-104.299522, -74.932858], [-103.772348, -74.698046], [-103.750383,
        -74.586316], [-103.974431, -74.559428], [-104.46646, -74.330813], [-104.690509,
        -74.214078], [-104.782764, -74.236776], [-104.558715, -74.424305], [-104.699295,
        -74.550065], [-105.164965, -74.595655]]
)
FI1fT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-109.906178, -73.254106], [-110.042365, -73.249041], [-110.130227,
        -73.263599], [-110.182944, -73.281306], [-110.160979, -73.309727], [-110.119244,
        -73.338731], [-110.073116, -73.35951], [-109.947913, -73.365171], [-109.807333,
        -73.365171], [-109.708488, -73.348808], [-109.717274, -73.31099], [-109.759009,
        -73.272455], [-109.906178, -73.254106]]
)
FI1T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([FI1aT,FI1bT,FI1cT,FI1dT,FI1eT,FI1fT])
)
RI1aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-109.064168, -73.803383], [-108.831334, -73.713662], [-108.330519,
        -73.833999], [-108.062539, -73.907249], [-108.365664, -73.913339], [-109.002665,
        -73.843785], [-109.064168, -73.803383]]
)
RI1bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-105.550729, -74.827975], [-105.910964, -74.837174], [-105.867033,
        -74.983609], [-105.818709, -75.055171], [-105.884605, -75.133171], [-105.998826,
        -75.229828], [-106.077902, -75.271226], [-106.490855, -75.281277], [-106.51282,
        -75.163583], [-106.561145, -75.055171], [-106.569931, -74.99954], [-106.798373,
        -74.984746], [-106.789586, -75.029085], [-106.829124, -75.153454], [-106.890628,
        -75.235428], [-107.004849, -75.326987], [-106.736869, -75.406919], [-106.108654,
        -75.412454], [-105.45408, -75.279045], [-104.724824, -75.109474], [-105.313501,
        -74.969937], [-105.550729, -74.827975]]
)
RI1cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-109.578517, -73.807057], [-109.877248, -73.887746], [-109.956324, -73.96319],
         [-109.912393, -74.045533], [-110.070545, -74.137081], [-109.815745, -74.137081],
        [-109.587303, -74.17547], [-109.517013, -74.230502], [-109.209495, -74.208993],
        [-108.963481, -74.19942], [-108.770184, -74.254371], [-108.480239, -74.24483],
        [-108.3133, -74.290107], [-108.067286, -74.24483], [-107.724623, -74.069674],
```

```
        [-107.891562, -73.997145], [-108.717467, -73.931594], [-109.112847, -73.877989],
        [-109.578517, -73.807057]]
)
RI1T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([RI1aT,RI1bT,RI1cT])
)
#######
OW2T = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-103.522965, -73.814411], [-103.426316, -73.815329], [-103.377992,
        -73.804914], [-103.340651, -73.793266], [-103.330766, -73.780076], [-103.522965,
        -73.814411]]
)
DW2aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-106.489038, -74.329625], [-106.910777, -74.387683], [-105.53134, -74.540696],
         [-105.329256, -74.492596], [-104.841621, -74.45024], [-104.859193, -74.290406],
        [-105.048097, -74.175775], [-105.360008, -74.138583], [-105.78614, -74.212879],
        [-105.799319, -74.291597], [-106.286955, -74.345048], [-106.489038, -74.329625]]
)
DW2bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-103.291228, -73.771788], [-103.312096, -73.776699], [-103.317587,
        -73.798784], [-103.286835, -73.798784], [-103.242904, -73.796638], [-103.240708,
        -73.784984], [-103.261575, -73.778848], [-103.291228, -73.771788]]
)
DW2T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([DW2aT,DW2bT])
)
FI2aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-104.511633, -74.550945], [-105.10031, -74.616366], [-104.432557, -74.787959],
         [-104.344695, -74.909709], [-104.195329, -75.025967], [-103.184914, -74.834014],
        [-102.824679, -74.861581], [-102.271146, -74.614032], [-103.017975, -74.447597],
        [-103.184914, -74.250801], [-104.019605, -74.326958], [-103.905384, -74.421652],
        [-103.606652, -74.445238], [-103.360638, -74.534549], [-104.010819, -74.618694],
        [-104.511633, -74.550945]]
)
FI2bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-102.359576, -74.126268], [-102.886749, -74.141887], [-102.741777,
        -74.255567], [-102.482583, -74.304382], [-102.293679, -74.265104], [-102.100382,
        -74.279398], [-101.81483, -74.274633], [-101.88512, -74.182663], [-102.183851,
        -74.129874], [-102.359576, -74.126268]]
)
FI2cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-106.602364, -74.696882], [-106.185019, -74.86301], [-106.180626, -74.814739],
         [-106.10155, -74.758233], [-106.044439, -74.675991], [-106.185019, -74.638783],
        [-106.563721, -74.664375], [-106.602364, -74.696882]]
)
FI2T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([FI2aT,FI2bT,FI2cT])
)
RI2T = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-105.768063, -74.835165], [-105.96136, -74.956546], [-105.205745, -75.290491],
         [-105.056379, -75.239073], [-104.028391, -75.18523], [-105.276034, -75.040719],
```

```
        [-105.504477, -74.890245], [-105.768063, -74.835165]]
)
###
OW3aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.099129, -74.210488], [-110.092539, -74.20092], [-110.046412, -74.199724],
         [-110.031036, -74.220052], [-110.068377, -74.224234], [-110.099129, -74.210488]]
)
OW3bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.150205, -74.383024], [-110.126043, -74.390712], [-110.11506, -74.371781],
         [-110.12165, -74.365861], [-110.141419, -74.363196], [-110.150205, -74.357272],
        [-110.168876, -74.362603], [-110.139222, -74.373557], [-110.150205, -74.383024]]
)
OW3cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-112.035948, -74.211758], [-112.026064, -74.197703], [-112.000803,
        -74.197703], [-111.966757, -74.203386], [-111.93271, -74.213551], [-112.003,
        -74.210862], [-112.035948, -74.211758]]
)
OW3T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([OW3aT,OW3bT,OW3cT])
)
DW3aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-113.363953, -74.145191], [-112.784062, -74.177574], [-112.520475,
        -74.190747], [-112.898283, -74.046442], [-112.819207, -73.98715], [-111.905439,
        -74.046442], [-110.908203, -73.939806], [-111.198149, -73.56481], [-113.271697,
        -73.936158], [-113.363953, -74.145191]]
)
DW3bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.448954, -74.118526], [-110.657627, -74.128745], [-110.870693,
        -74.138657], [-111.017862, -74.220424], [-110.875086, -74.277089], [-110.727917,
        -74.237445], [-110.777339, -74.222217], [-110.765258, -74.172552], [-110.708148,
        -74.166258], [-110.660922, -74.183336], [-110.654332, -74.227592], [-110.463232,
        -74.244307], [-110.381959, -74.27292], [-110.514851, -74.200693], [-110.312768,
        -74.165359], [-110.339127, -74.158762], [-110.329242, -74.147062], [-110.340225,
        -74.136555], [-110.426989, -74.135354], [-110.448954, -74.118526]]
)
DW3cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.098604, -74.299474], [-110.094211, -74.288173], [-110.130454,
        -74.284602], [-110.149124, -74.275673], [-110.195252, -74.272993], [-110.199645,
        -74.258097], [-110.255657, -74.252134], [-110.26664, -74.266441], [-110.251264,
        -74.282818], [-110.207333, -74.2849], [-110.20953, -74.292635], [-110.318259,
        -74.287282], [-110.340225, -74.310769], [-110.328144, -74.324429], [-110.294097,
        -74.344898], [-110.302883, -74.32799], [-110.263345, -74.318491], [-110.218316,
        -74.324725], [-110.19635, -74.318787], [-110.20953, -74.307797], [-110.135945,
        -74.292933], [-110.124962, -74.301258], [-110.098604, -74.299474]]
)
DW3T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([DW3aT,DW3bT,DW3cT])
)
FI3aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-107.834207, -74.698189], [-107.950624, -74.703986], [-107.937445,
        -74.738148], [-107.821027, -74.752601], [-107.752934, -74.772235], [-107.741951,
        -74.795303], [-107.684841, -74.757223], [-107.834207, -74.698189]]
)
```

```
FI3bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-109.035723, -74.604696], [-109.112602, -74.615777], [-109.099423, -74.62976],
         [-109.196071, -74.641403], [-109.165319, -74.665828], [-109.092833, -74.651295],
        [-109.051098, -74.656528], [-108.888553, -74.643732], [-108.921502, -74.620439],
        [-108.991792, -74.614611], [-109.035723, -74.604696]]
)
FI3cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-109.833072, -74.553429], [-109.995618, -74.555183], [-110.063711,
        -74.600177], [-109.971455, -74.603677], [-109.804517, -74.622333], [-109.699083,
        -74.585583], [-109.61122, -74.566882], [-109.433299, -74.54816], [-109.580468,
        -74.507715], [-109.657348, -74.529416], [-109.743014, -74.521794], [-109.795731,
        -74.539376], [-109.714458, -74.549916], [-109.789141, -74.584998], [-109.877004,
        -74.579742], [-109.833072, -74.553429]]
)
FI3dT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-108.780922, -74.367562], [-108.875374, -74.373482], [-108.789708,
        -74.409546], [-108.743581, -74.444349], [-108.706239, -74.437277], [-108.62277,
        -74.45142], [-108.581036, -74.446707], [-108.695257, -74.402458], [-108.780922,
        -74.367562]]
)
FI3eT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.142535, -74.142711], [-110.179876, -74.145712], [-110.161205,
        -74.171204], [-110.077736, -74.187078], [-110.060164, -74.175998], [-110.014036,
        -74.173301], [-109.9734, -74.187378], [-109.910798, -74.196356], [-109.880046,
        -74.192167], [-109.905307, -74.166708], [-110.01184, -74.153214], [-110.07554,
        -74.152614], [-110.142535, -74.142711]]
)
FI3fT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.093112, -74.373999], [-110.1008, -74.396178], [-110.098604, -74.412128],
        [-110.076638, -74.406222], [-110.073343, -74.389675], [-110.077736, -74.37903],
        [-110.071147, -74.373112], [-110.093112, -74.373999]]
)
FI3T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([FI3aT,FI3bT,FI3cT,FI3dT,FI3eT,FI3fT])
)
RI3aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.220444, -73.854484], [-110.189692, -74.025896], [-110.378596,
        -74.071789], [-110.361023, -74.119958], [-110.040327, -74.127172], [-109.811885,
        -74.130775], [-109.76356, -74.19793], [-109.631767, -74.20511], [-109.403325,
        -74.172778], [-110.220444, -73.854484]]
)
RI3bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-112.262731, -74.260926], [-112.456028, -74.239457], [-112.710829,
        -74.232292], [-112.912912, -74.208392], [-113.194071, -74.215567], [-113.431299,
        -74.246618], [-113.457658, -74.640238], [-113.369795, -74.889669], [-112.412097,
        -74.983317], [-112.719615, -75.195897], [-111.524689, -75.305537], [-111.111736,
        -75.184665], [-110.083748, -75.17792], [-109.600506, -75.089975], [-108.98547,
        -74.869032], [-109.328133, -74.800053], [-109.249057, -74.698332], [-109.52143,
        -74.642563], [-109.934382, -74.656525], [-110.074962, -74.850666], [-111.03266,
        -74.95825], [-111.840993, -74.898829], [-112.095793, -74.716877], [-112.051862,
        -74.258545], [-112.262731, -74.260926]]
)
RI3cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
```

```
    coords = [[-108.445819, -74.906555], [-108.705012, -74.929424], [-108.612757, -75.00238],
         [-108.454605, -75.063666], [-108.248129, -75.076121], [-108.147087, -75.019429],
        [-108.072404, -74.979619], [-108.147087, -74.921425], [-108.445819, -74.906555]]
)
RI3T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([RI3aT,RI3bT,RI3cT])
)
###
OW4aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-111.542252, -73.671708], [-111.357741, -73.629649], [-111.300631,
        -73.708731], [-111.107334, -73.759198], [-111.098548, -73.822985], [-111.019472,
        -73.880431], [-110.927217, -73.974112], [-111.094155, -74.029825], [-111.089762,
        -74.097395], [-111.239127, -74.046744], [-111.072189, -73.924296], [-111.203982,
        -73.919428], [-111.300631, -73.88165], [-111.199589, -73.85601], [-111.340169,
        -73.821761], [-111.296238, -73.787441], [-111.423638, -73.778851], [-111.533466,
        -73.719821], [-111.542252, -73.671708]]
)
OW4bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-111.83944, -73.665835], [-111.894354, -73.680968], [-111.840538, -73.682202],
         [-111.815278, -73.690535], [-111.851521, -73.726593], [-111.833949, -73.751819],
        [-111.852619, -73.769024], [-111.828457, -73.788358], [-111.797705, -73.808897],
        [-111.758167, -73.820533], [-111.749381, -73.834303], [-111.691172, -73.830021],
        [-111.676895, -73.858145], [-111.656027, -73.830633], [-111.618686, -73.823289],
        [-111.627472, -73.789892], [-111.617588, -73.758887], [-111.654929, -73.744132],
        [-111.670305, -73.719819], [-111.723022, -73.704413], [-111.83944, -73.665835]]
)
OW4cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-109.112746, -74.652494], [-109.112197, -74.655401], [-109.102861, -74.657],
        [-109.086936, -74.654966], [-109.112746, -74.652494]]
)
OW4T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([OW4aT,OW4bT,OW4cT])
)
DW4aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-112.456034, -73.784373], [-113.244597, -73.908467], [-113.007369,
        -74.031637], [-112.778928, -73.93281], [-112.379155, -73.889577], [-112.346206,
        -73.920643], [-112.653724, -74.010467], [-113.242401, -74.081733], [-113.19188,
        -74.160488], [-112.752569, -74.108829], [-112.651527, -74.067264], [-112.304472,
        -74.011677], [-112.238575, -74.039494], [-112.93708, -74.180266], [-111.922271,
        -74.185058], [-111.884929, -74.115446], [-112.113371, -74.08113], [-111.994757,
        -73.896282], [-112.456034, -73.784373]]
)
DW4bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-112.487978, -73.787131], [-113.226021, -73.910598], [-112.9844, -74.035565],
        [-112.791103, -73.923987], [-112.417689, -73.894757], [-112.338612, -73.920338],
        [-112.628558, -74.008951], [-113.243594, -74.080225], [-113.18209, -74.155988],
        [-112.760351, -74.110332], [-112.690061, -74.069371], [-112.329826, -74.013794],
        [-112.255143, -74.035565], [-112.958041, -74.178767], [-112.246357, -74.189545],
        [-112.211212, -74.075401], [-112.140922, -73.970165], [-112.022308, -73.897194],
        [-112.487978, -73.787131]]
)
DW4cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.005993, -74.182512], [-110.111428, -74.18311], [-110.07848, -74.213626],
        [-110.069694, -74.242294], [-110.027959, -74.264357], [-110.034549, -74.285795],
        [-109.94449, -74.284604], [-109.896166, -74.269123], [-109.753389, -74.263165],
```

```
        [-109.713851, -74.240503], [-109.700672, -74.217809], [-109.716048, -74.191494],
        [-109.762176, -74.166334], [-109.957669, -74.199871], [-110.005993, -74.182512]]
)
DW4dT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.658371, -73.950131], [-110.724267, -73.957419], [-110.588081,
        -74.031335], [-110.487039, -74.01319], [-110.423339, -74.028312], [-110.423339,
        -74.075402], [-110.390391, -74.122961], [-110.39698, -74.154789], [-110.616636,
        -74.206301], [-110.618832, -74.227219], [-110.480449, -74.242741], [-110.344263,
        -74.217661], [-110.377211, -74.19254], [-110.322297, -74.140984], [-110.225649,
        -74.120555], [-110.201487, -74.085046], [-110.096052, -74.110332], [-110.089463,
        -74.09107], [-110.170735, -74.078416], [-110.225649, -74.047648], [-110.258597,
        -74.02166], [-110.26958, -73.970774], [-110.429929, -73.990783], [-110.53756,
        -73.988966], [-110.59467, -73.951346], [-110.658371, -73.950131]]
)
DW4eT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.02278, -74.409028], [-110.018387, -74.414638], [-109.995323, -74.41021],
        [-109.963473, -74.407552], [-109.948097, -74.403122], [-109.922837, -74.403713],
        [-109.940409, -74.3981], [-109.967866, -74.399577], [-109.982144, -74.404599],
        [-110.001913, -74.405485], [-110.02278, -74.409028]]
)
DW4fT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.132608, -74.361048], [-110.180932, -74.363418], [-110.202897,
        -74.372299], [-110.228158, -74.385315], [-110.216077, -74.395366], [-110.109544,
        -74.397139], [-110.131509, -74.380879], [-110.132608, -74.361048]]
)
DW4gT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.035959, -74.306313], [-110.018387, -74.316116], [-110.013994,
        -74.331849], [-110.059023, -74.322352], [-110.066711, -74.311364], [-110.035959,
        -74.306313]]
)
DW4hT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.34787, -74.311289], [-110.27758, -74.348086], [-110.251222, -74.338301],
        [-110.264401, -74.331478], [-110.217175, -74.325543], [-110.11174, -74.302969],
        [-110.167753, -74.272622], [-110.24573, -74.274111], [-110.283072, -74.287505],
        [-110.242435, -74.296428], [-110.273187, -74.304456], [-110.308332, -74.299105],
        [-110.337986, -74.302078], [-110.34787, -74.311289]]
)
DW4T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([DW4aT,DW4bT,DW4cT,DW4dT,DW4eT,DW4fT,DW4gT,DW4hT])
)
FI4aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-109.630382, -74.133626], [-109.74021, -74.135429], [-109.665527, -74.204358],
         [-109.623793, -74.197778], [-109.590844, -74.203162], [-109.608417, -74.22707],
        [-109.654544, -74.246171], [-109.738014, -74.281331], [-109.647955, -74.2968],
        [-109.593041, -74.316414], [-109.520555, -74.272995], [-109.562289, -74.255117],
        [-109.496392, -74.238414], [-109.452461, -74.221694], [-109.404137, -74.216914],
        [-109.347027, -74.257503], [-109.28113, -74.243785], [-109.206447, -74.227668],
        [-109.27454, -74.200769], [-109.360206, -74.210936], [-109.397547, -74.206152],
        [-109.34483, -74.187601], [-109.434889, -74.173826], [-109.388761, -74.156438],
        [-109.441478, -74.133026], [-109.496392, -74.141434], [-109.437085, -74.151638],
        [-109.430496, -74.163036], [-109.520555, -74.180415], [-109.608417, -74.181015],
        [-109.645758, -74.164835], [-109.634775, -74.154039], [-109.575469, -74.154638],
        [-109.575469, -74.139033], [-109.630382, -74.133626]]
)
FI4bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
```

```
    coords = [[-109.803011, -74.523262], [-109.967753, -74.526778], [-110.057812,
        -74.573606], [-110.051222, -74.59113], [-109.978736, -74.584123], [-109.866711,
        -74.596967], [-109.750294, -74.593466], [-109.653645, -74.555477], [-109.423007,
        -74.51681], [-109.335145, -74.488043], [-109.392255, -74.465107], [-109.541621,
        -74.457457], [-109.655842, -74.456867], [-109.603125, -74.492743], [-109.651449,
        -74.505662], [-109.697577, -74.497441], [-109.811797, -74.507423], [-109.671218,
        -74.523262], [-109.776653, -74.563083], [-109.833763, -74.557233], [-109.803011,
        -74.523262]]
)
FI4cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-108.824854, -74.358901], [-108.856704, -74.37548], [-108.810576, -74.38465],
        [-108.807281, -74.399134], [-108.913814, -74.41331], [-108.873178, -74.427769],
        [-108.775431, -74.428063], [-108.688667, -74.43455], [-108.698552, -74.443392],
        [-108.611788, -74.442213], [-108.824854, -74.358901]]
)
FI4dT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-104.002142, -73.351324], [-104.085611, -73.429207], [-103.793469,
        -73.445492], [-103.644104, -73.412279], [-104.002142, -73.351324]]
)
FI4T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([FI4aT,FI4bT,FI4cT,FI4dT])
)
RI4aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.506535, -73.745365], [-110.616363, -73.735521], [-110.664687,
        -73.789587], [-110.493356, -73.801851], [-110.528501, -73.847147], [-110.598791,
        -73.843481], [-110.677867, -73.877682], [-110.594397, -73.866697], [-110.506535,
        -73.913035], [-110.431852, -73.913035], [-110.427459, -73.965315], [-110.199017,
        -73.939803], [-110.155086, -74.033149], [-109.926644, -74.089866], [-109.737741,
        -74.048857], [-110.506535, -73.745365]]
)
RI4bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-112.241112, -74.254365], [-112.645278, -74.259133], [-112.970368,
        -74.242435], [-113.242741, -74.204198], [-113.427252, -74.228105], [-113.462397,
        -74.354226], [-113.532686, -74.519293], [-113.576618, -74.777567], [-114.13015,
        -74.965655], [-112.987941, -74.913127], [-112.539843, -75.038437], [-112.451981,
        -75.180724], [-111.142834, -75.232341], [-110.026983, -75.203188], [-109.245009,
        -75.221135], [-107.821642, -74.974769], [-108.164304, -74.876479], [-108.919919,
        -75.092796], [-109.491024, -75.137949], [-109.578886, -75.036165], [-108.867202,
        -74.929131], [-108.946278, -74.82135], [-109.3153, -74.749845], [-109.271368,
        -74.638488], [-109.912762, -74.629168], [-110.439936, -74.91999], [-111.547,
        -74.922273], [-112.232325, -74.661753], [-112.074173, -74.448759], [-112.241112,
        -74.254365]]
)
RI4cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-108.623786, -74.584562], [-108.887372, -74.60674], [-108.975234, -74.626557],
         [-108.935697, -74.676575], [-108.830262, -74.691665], [-108.641358, -74.675415],
        [-108.579854, -74.637041], [-108.623786, -74.584562]]
)
RI4T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([RI4aT,RI4bT,RI4cT])
)
#######
OW5T = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-105.683101, -74.317753], [-105.549111, -74.35809], [-105.393155, -74.358683],
         [-105.305293, -74.330814], [-105.199858, -74.299332], [-105.208644, -74.276721],
```

```
        [-105.162517, -74.236777], [-105.076851, -74.229015], [-104.993382, -74.242745],
        [-104.912109, -74.27136], [-104.868178, -74.304682], [-104.876965, -74.359867],
        [-104.826444, -74.380587], [-104.782513, -74.360459], [-104.797889, -74.316566],
        [-104.833033, -74.29041], [-104.868178, -74.254078], [-104.916503, -74.214675],
        [-104.971417, -74.176376], [-105.070262, -74.163788], [-105.177893, -74.156592],
        [-105.149338, -74.127772], [-105.235003, -74.123565], [-105.276738, -74.139786],
        [-105.360207, -74.148791], [-105.441479, -74.162589], [-105.564487, -74.197931],
        [-105.700673, -74.234986], [-105.727032, -74.289219], [-105.683101, -74.317753]]
)
DW5aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-105.733919, -74.005019], [-105.949182, -74.072692], [-106.107334, -74.15569],
         [-106.322596, -74.251392], [-106.480748, -74.302599], [-106.90688, -74.42224],
        [-106.915666, -74.471728], [-106.985956, -74.558548], [-106.665259, -74.642568],
        [-106.515893, -74.639079], [-106.388493, -74.648387], [-106.274272, -74.524582],
        [-106.006292, -74.510505], [-105.483512, -74.556209], [-105.281429, -74.536303],
        [-105.272643, -74.506983], [-104.991484, -74.519889], [-104.815759, -74.487015],
        [-104.872869, -74.430498], [-105.417615, -74.437572], [-106.195196, -74.372595],
        [-106.107334, -74.301409], [-105.874499, -74.202417], [-105.378077, -74.10522],
        [-105.430795, -74.066662], [-105.703167, -74.069074], [-105.698774, -74.042516],
        [-105.641664, -74.019543], [-105.733919, -74.005019]]
)
DW5bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-107.443372, -73.962811], [-107.510367, -73.993134], [-107.430192,
        -74.006156], [-107.414816, -73.988286], [-107.384065, -73.987983], [-107.366492,
        -74.002825], [-107.325856, -74.011906], [-107.337937, -74.029448], [-107.320364,
        -74.037305], [-107.331347, -74.047876], [-107.325856, -74.070808], [-107.296202,
        -74.056932], [-107.304989, -74.045461], [-107.297301, -74.035794], [-107.24678,
        -74.031261], [-107.200652, -74.026425], [-107.15123, -74.027634], [-107.077645,
        -74.046367], [-107.11938, -74.019167], [-107.195161, -74.020982], [-107.279728,
        -74.023099], [-107.294006, -74.010998], [-107.304989, -73.979192], [-107.357706,
        -73.97525], [-107.39944, -73.964935], [-107.443372, -73.962811]]
)
DW5cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-107.064764, -74.095024], [-107.049937, -74.096228], [-107.04829, -74.09111],
        [-107.037856, -74.087495], [-107.042798, -74.082525], [-107.044995, -74.078457],
        [-107.045544, -74.073031], [-107.055978, -74.080415], [-107.046093, -74.085387],
        [-107.052134, -74.088399], [-107.064764, -74.095024]]
)
DW5T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([DW5aT,DW5bT,DW5cT])
)
FI5aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-104.582923, -74.242744], [-104.727896, -74.259444], [-104.644427,
        -74.424305], [-104.736682, -74.537184], [-105.145242, -74.561768], [-105.386863,
        -74.576968], [-105.048593, -74.669028], [-104.951945, -74.782479], [-104.429164,
        -74.842348], [-104.71911, -75.027384], [-103.919564, -75.071598], [-104.011819,
        -75.026247], [-103.550542, -74.986455], [-103.511004, -74.92943], [-103.076086,
        -74.807838], [-103.770198, -74.550065], [-103.906384, -74.559428], [-104.165578,
        -74.506691], [-104.411592, -74.531325], [-104.446737, -74.503168], [-104.064536,
        -74.454957], [-104.152398, -74.405416], [-104.288585, -74.398325], [-104.288585,
        -74.356905], [-104.582923, -74.242744]]
)
FI5bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-107.188039, -74.564401], [-107.394515, -74.649554], [-107.319832,
        -74.713407], [-107.126535, -74.663507], [-107.016707, -74.679773], [-107.043066,
        -74.750438], [-106.920059, -74.834589], [-107.047459, -74.90456], [-106.902487,
        -74.910281], [-106.775086, -74.833439], [-106.62572, -74.842634], [-106.573003,
        -74.933145], [-106.49832, -74.898837], [-106.458782, -74.85527], [-106.195195,
        -74.842634], [-106.199589, -74.815034], [-106.037043, -74.694857], [-106.195195,
```

```
        -74.644901], [-106.388492, -74.700656], [-106.502713, -74.682094], [-106.647686,
        -74.669317], [-106.82341, -74.707613], [-106.90688, -74.698337], [-106.911273,
        -74.635588], [-107.012314, -74.595948], [-107.188039, -74.564401]]
)
FI5cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-107.313711, -73.77041], [-107.493829, -73.812729], [-107.392787, -73.886073],
         [-107.159953, -73.926275], [-107.058911, -73.993664], [-106.88758, -73.9973],
        [-106.845845, -73.964556], [-106.540524, -73.982149], [-106.465841, -73.945122],
        [-106.503182, -73.864103], [-106.531738, -73.790662], [-106.716248, -73.792503],
        [-106.832666, -73.763653], [-106.643762, -73.698404], [-106.623993, -73.637234],
        [-106.751393, -73.637853], [-106.779948, -73.709502], [-106.911742, -73.738445],
        [-106.971049, -73.72552], [-106.946887, -73.667537], [-107.089663, -73.694086],
        [-107.036945, -73.734138], [-107.113825, -73.75628], [-107.197294, -73.724903],
        [-107.144577, -73.668774], [-107.326891, -73.66692], [-107.408163, -73.705804],
        [-107.320301, -73.742753], [-107.313711, -73.77041]]
)
FI5dT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-108.269213, -73.83905], [-108.576731, -73.89522], [-108.458116, -73.884855],
        [-108.214299, -73.903142], [-108.093488, -73.918974], [-108.084702, -73.89583],
        [-107.99684, -73.907405], [-107.959498, -73.894611], [-108.135223, -73.865327],
        [-108.190137, -73.873264], [-108.240657, -73.863495], [-108.18794, -73.846387],
        [-108.269213, -73.83905]]
)
FI5eT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-107.199489, -74.036628], [-107.269779, -74.04146], [-107.25001, -74.061988],
        [-107.298334, -74.086105], [-107.291745, -74.135433], [-107.18631, -74.109584],
        [-107.096251, -74.094537], [-107.0655, -74.060782], [-107.148969, -74.040857],
        [-107.199489, -74.036628]]
)
FI5T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([FI5aT,FI5bT,FI5cT,FI5dT,FI5eT])
)
RI5aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.850815, -74.921715], [-110.23578, -75.189163], [-109.585599, -75.175674],
         [-108.627901, -75.211614], [-108.276452, -75.312231], [-107.125457, -75.292158],
        [-106.571925, -75.405535], [-104.313865, -75.108062], [-105.236419, -75.031073],
        [-105.596654, -74.832285], [-105.93053, -74.813881], [-106.07111, -75.083193],
        [-106.633428, -74.996982], [-107.055167, -75.144162], [-107.32754, -75.074143],
        [-107.353899, -75.001531], [-107.591126, -75.008354], [-107.79321, -75.13965],
        [-108.179804, -75.13289], [-108.241307, -75.060552], [-107.951361, -74.907988],
        [-108.452177, -74.832285], [-108.803625, -75.087722], [-109.260509, -75.148667],
        [-109.295654, -75.092244], [-109.172647, -74.981043], [-108.926632, -74.864449],
        [-109.286868, -74.81618], [-109.357157, -74.737717], [-109.269295, -74.640244],
        [-109.401089, -74.595946], [-109.550454, -74.721515], [-109.69982, -74.658853],
        [-109.954621, -74.647222], [-109.708606, -74.742343], [-110.007338, -74.751589],
        [-110.191849, -74.772383], [-110.174276, -74.850672], [-110.850815, -74.921715]]
)
RI5bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.042483, -74.156893], [-108.680618, -73.939503], [-107.79321, -74.00744],
        [-107.801996, -74.120862], [-108.1183, -74.293087], [-108.900274, -74.25497],
        [-109.084785, -74.209595], [-109.488951, -74.24543], [-109.655889, -74.159289],
        [-110.042483, -74.156893]]
)
RI5T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([RI5aT,RI5bT])
)
###
```

```
OW6aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.647432, -73.095821], [-110.625467, -73.088153], [-110.592518,
        -73.079521], [-110.599108, -73.075044], [-110.579339, -73.071206], [-110.59801,
        -73.069926], [-110.628762, -73.076004], [-110.649629, -73.084637], [-110.647432,
        -73.095821]]
)
OW6bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.621074, -73.101171], [-110.649629, -73.105323], [-110.628762,
        -73.114898], [-110.633155, -73.124787], [-110.610091, -73.118089], [-110.607894,
        -73.110749], [-110.621074, -73.101171]]
)
OW6T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([OW6aT,OW6bT])
)
DW6aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-111.641307, -73.088549], [-113.337048, -73.555167], [-112.309059,
        -74.091374], [-111.694024, -74.072089], [-111.483155, -74.031035], [-110.98234,
        -74.067265], [-110.815401, -73.953477], [-111.325002, -73.639555], [-111.078988,
        -73.497848], [-111.281071, -73.390155], [-111.316216, -73.238746], [-111.456796,
        -73.032213], [-111.641307, -73.088549]]
)
DW6bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.346504, -72.879498], [-110.302573, -72.880145], [-110.246561,
        -72.885965], [-110.19604, -72.890168], [-110.152109, -72.899216], [-110.089507,
        -72.893076], [-110.097195, -72.882732], [-110.059853, -72.879822], [-110.06864,
        -72.872058], [-110.048871, -72.863644], [-110.020315, -72.86073], [-109.9665,
        -72.858139], [-109.956615, -72.851014], [-109.939043, -72.845182], [-109.932453,
        -72.838376], [-109.981876, -72.846479], [-109.995055, -72.854901], [-110.075229,
        -72.865909], [-110.067541, -72.857492], [-110.090605, -72.859435], [-110.094998,
        -72.870117], [-110.168583, -72.872381], [-110.210318, -72.87691], [-110.26633,
        -72.874322], [-110.304769, -72.874646], [-110.346504, -72.879498]]
)
DW6cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-108.922311, -73.030451], [-108.900345, -73.032375], [-108.861906, -73.02147],
         [-108.870692, -73.00478], [-108.891559, -73.00478], [-108.911328, -72.997392],
        [-108.911328, -72.99193], [-108.859709, -72.989681], [-108.905837, -72.981966],
        [-108.944276, -73.000605], [-108.978323, -73.00221], [-109.050809, -73.005743],
        [-109.072775, -73.003174], [-109.111215, -73.007349], [-109.071677, -73.010559],
        [-109.050809, -73.02147], [-109.022254, -73.008312], [-108.993699, -73.009275],
        [-108.936588, -73.014411], [-108.921212, -73.022433], [-108.922311, -73.030451]]
)
DW6dT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-109.256187, -73.009516], [-109.233123, -73.004058], [-109.243008,
        -72.996991], [-109.22873, -72.992493], [-109.211158, -72.997634], [-109.193585,
        -72.994099], [-109.173816, -72.995706], [-109.188094, -73.001809], [-109.194684,
        -73.009516], [-109.221042, -73.01401], [-109.256187, -73.009516]]
)
DW6eT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-111.713487, -74.195908], [-111.731059, -74.193215], [-111.732706, -74.18648],
         [-111.71129, -74.185133], [-111.721724, -74.17465], [-111.714585, -74.169406],
        [-111.706897, -74.167607], [-111.651434, -74.16506], [-111.64759, -74.168207],
        [-111.671203, -74.172103], [-111.692619, -74.175099], [-111.701406, -74.185582],
        [-111.715134, -74.189324], [-111.713487, -74.195908]]
)
DW6fT = ee.Geometry.LineString(
```

```
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-107.836978, -73.822372], [-107.86114, -73.831859], [-107.834781, -73.851426],
         [-107.780966, -73.851426], [-107.739231, -73.883784], [-107.705185, -73.891407],
        [-107.68871, -73.915165], [-107.684317, -73.880124], [-107.739231, -73.849287],
        [-107.697497, -73.837364], [-107.641484, -73.837059], [-107.606339, -73.860896],
        [-107.59975, -73.879819], [-107.607438, -73.900244], [-107.552524, -73.904508],
        [-107.508593, -73.94193], [-107.481136, -73.973808], [-107.42073, -73.99442],
        [-107.352637, -74.000478], [-107.22963, -73.985936], [-107.301018, -73.9641],
        [-107.297723, -73.937371], [-107.353736, -73.938283], [-107.386684, -73.959547],
        [-107.4405, -73.95317], [-107.453679, -73.940715], [-107.391077, -73.92977],
        [-107.376799, -73.879819], [-107.449286, -73.866087], [-107.467956, -73.892016],
        [-107.56131, -73.884699], [-107.562408, -73.874938], [-107.486627, -73.843785],
        [-107.563507, -73.838893], [-107.579981, -73.822066], [-107.605241, -73.794801],
        [-107.646976, -73.804916], [-107.643681, -73.814718], [-107.773278, -73.831553],
        [-107.836978, -73.822372]]
)
DW6gT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-106.792938, -73.671474], [-106.79184, -73.682898], [-106.820395, -73.690922],
         [-106.814903, -73.697091], [-106.764383, -73.681664], [-106.764383, -73.677033],
        [-106.732533, -73.668076], [-106.731434, -73.662205], [-106.669931, -73.658496],
        [-106.641375, -73.652004], [-106.606231, -73.650767], [-106.598543, -73.643343],
        [-106.645769, -73.643962], [-106.665538, -73.642724], [-106.678717, -73.650457],
        [-106.699584, -73.652004], [-106.729238, -73.656951], [-106.752301, -73.672092],
        [-106.767677, -73.672709], [-106.792938, -73.671474]]
)
DW6T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([DW6aT,DW6bT,DW6cT,DW6dT,DW6eT,DW6fT,DW6gT])
)
FI6T = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-108.298148, -73.138021], [-108.3289, -73.432499], [-108.175141, -73.473801],
        [-107.907161, -73.48005], [-107.718257, -73.539925], [-107.53814, -73.55486],
        [-107.749009, -73.490044], [-107.626002, -73.473801], [-107.46785, -73.506272],
        [-107.459064, -73.47005], [-107.450278, -73.389842], [-107.718257, -73.413693],
        [-107.766582, -73.388585], [-107.687506, -73.379789], [-107.353629, -73.358408],
        [-107.305305, -73.325654], [-107.489816, -73.315566], [-107.617216, -73.33952],
        [-107.718257, -73.342039], [-107.683113, -73.297892], [-107.766582, -73.268818],
        [-107.573285, -73.242232], [-107.463457, -73.228288], [-107.388774, -73.173674],
        [-107.336057, -73.109952], [-107.511781, -73.070329], [-107.674326, -73.038308],
        [-107.841265, -73.043434], [-107.80612, -73.157131], [-107.955485, -73.168586],
        [-108.069706, -73.106124], [-108.298148, -73.138021]]
)
RI6T = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-108.689137, -74.223929], [-107.898376, -74.09438], [-107.951094, -73.985629],
         [-108.61006, -73.922463], [-109.418393, -73.84928], [-109.804987, -73.765951],
        [-110.14765, -73.844391], [-110.14765, -73.963794], [-110.279443, -74.091968],
        [-109.892849, -74.11123], [-109.655621, -74.197626], [-109.233882, -74.144886],
        [-108.689137, -74.223929]]
)
####
OW7aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-102.891684, -73.380105], [-102.913649, -73.406478], [-102.891684,
        -73.449097], [-102.658849, -73.44534], [-102.610525, -73.486611], [-102.557807,
        -73.415261], [-102.755497, -73.387643], [-102.891684, -73.380105]]
)
OW7bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-142.818282, -74.217513], [-142.805103, -74.228864], [-142.763368,
        -74.214525], [-142.719437, -74.20376], [-142.704061, -74.19419], [-142.646951,
        -74.176822], [-142.60302, -74.160639], [-142.690882, -74.175625], [-142.73701,
```

```
        -74.185806], [-142.758975, -74.200769], [-142.798513, -74.208546], [-142.818282,
        -74.217513]]
)
OW7cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-137.465611, -74.898116], [-137.75336, -74.915853], [-137.731394, -74.885515],
         [-137.639139, -74.8752], [-137.575439, -74.884371], [-137.472201, -74.887809],
        [-137.465611, -74.898116]]
)
OW7T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([OW7aT,OW7bT,OW7cT])
)
DW7T = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-137.776459, -75.017583], [-137.797326, -75.029791], [-137.732528,
        -75.041706], [-137.741314, -75.055311], [-137.62929, -75.040855], [-137.655648,
        -75.032062], [-137.65455, -75.020423], [-137.594145, -75.020139], [-137.555705,
        -75.022694], [-137.526052, -75.018719], [-137.460155, -75.019571], [-137.420617,
        -75.024114], [-137.373391, -75.011333], [-137.374489, -75.000817], [-137.378882,
        -74.992], [-137.337148, -74.987162], [-137.281136, -74.983748], [-137.244892,
        -74.986024], [-137.169111, -74.986877], [-137.176799, -74.976914], [-137.153735,
        -74.974636], [-137.106509, -74.982324], [-137.057087, -74.995698], [-137.030728,
        -75.017868], [-136.999976, -75.019288], [-136.948357, -74.997689], [-137.038416,
        -74.987162], [-137.116394, -74.971502], [-137.08674, -74.963524], [-137.071365,
        -74.94784], [-137.030728, -74.938992], [-136.970323, -74.929569], [-136.925294,
        -74.928141], [-136.884657, -74.928141], [-136.8572, -74.935281], [-136.830842,
        -74.92414], [-136.802286, -74.897543], [-136.835235, -74.8752], [-136.891247,
        -74.874913], [-137.031827, -74.876059], [-137.108706, -74.876633], [-137.241598,
        -74.874913], [-137.339344, -74.879785], [-137.385472, -74.884943], [-137.38657,
        -74.894107], [-137.334951, -74.895825], [-137.304199, -74.901836], [-137.245991,
        -74.908702], [-137.19547, -74.909275], [-137.092232, -74.908702], [-137.05489,
        -74.908988], [-137.053792, -74.916994], [-137.081249, -74.929283], [-137.09992,
        -74.935566], [-137.154834, -74.936709], [-137.204256, -74.942133], [-137.235008,
        -74.950122], [-137.241598, -74.959248], [-137.282234, -74.970648], [-137.332755,
        -74.975206], [-137.421715, -74.975206], [-137.485415, -74.974636], [-137.523855,
        -74.978337], [-137.467843, -74.979192], [-137.419519, -74.98204], [-137.414027,
        -74.988585], [-137.45686, -74.989723], [-137.561196, -74.99456], [-137.657845,
        -74.998826], [-137.710562, -75.009628], [-137.776459, -75.017583]]
)
FI7aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-141.567056, -74.353642], [-141.865788, -74.40098], [-141.716422, -74.429314],
         [-141.716422, -74.476429], [-141.791105, -74.545669], [-141.764746, -74.586601],
        [-141.606594, -74.586601], [-141.479194, -74.557374], [-141.316649, -74.574917],
        [-141.250752, -74.613445], [-141.220001, -74.569072], [-141.000345, -74.570242],
        [-140.754331, -74.58777], [-140.44242, -74.590108], [-140.16126, -74.608782],
        [-139.924033, -74.595946], [-139.928426, -74.555037], [-140.143688, -74.550357],
        [-140.213978, -74.50111], [-140.389702, -74.464664], [-140.539068, -74.522231],
        [-140.745544, -74.524579], [-140.833407, -74.498759], [-140.978379, -74.48701],
        [-141.189249, -74.476429], [-141.206821, -74.454068], [-141.048669, -74.432856],
        [-141.118959, -74.39507], [-141.395725, -74.390342], [-141.457229, -74.359567],
        [-141.567056, -74.353642]]
)
FI7bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-138.123355, -74.839469], [-138.093702, -74.846077], [-138.071736,
        -74.835159], [-138.032198, -74.823371], [-138.002545, -74.813014], [-137.995955,
        -74.794583], [-138.002545, -74.775842], [-138.005839, -74.758811], [-138.023412,
        -74.749278], [-138.06295, -74.756212], [-138.081621, -74.76603], [-138.096997,
        -74.780168], [-138.103586, -74.80956], [-138.123355, -74.839469]]
)
FI7cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-140.046506, -74.81244], [-140.329861, -74.820496], [-140.22223, -74.903124],
```

```
        [-140.444082, -75.023406], [-140.613217, -75.028515], [-140.755993, -74.970505],
        [-140.911949, -74.923141], [-140.868017, -74.851533], [-141.021776, -74.846937],
        [-141.201894, -74.890528], [-141.133801, -74.956824], [-140.907555, -75.006361],
        [-140.720848, -75.111166], [-140.641772, -75.146128], [-140.488013, -75.081779],
        [-140.187085, -75.085173], [-140.235409, -75.051203], [-139.868585, -75.010907],
        [-139.840029, -74.971075], [-139.82026, -74.883655], [-140.046506, -74.81244]]
)
FI7dT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-139.165965, -74.96851], [-139.291169, -75.029365], [-139.227469, -75.07584],
        [-139.025386, -75.058854], [-138.904575, -75.10467], [-138.673937, -75.113705],
        [-138.658561, -75.08998], [-138.834285, -75.086021], [-138.840875, -75.066215],
        [-138.665151, -75.066215], [-138.260984, -75.07584], [-138.307112, -75.058287],
        [-138.47405, -75.046384], [-138.673937, -75.024257], [-138.636595, -75.012897],
        [-138.445495, -75.011759], [-138.203874, -75.008348], [-138.083063, -75.015169],
        [-138.206071, -74.985596], [-138.355436, -74.971359], [-138.533357, -74.9725],
        [-138.649775, -74.986166], [-138.766192, -75.008348], [-138.873823, -75.004371],
        [-138.904575, -74.98389], [-139.165965, -74.96851]]
)
FI7T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([FI7aT,FI7bT,FI7cT,FI7dT])
)
RI7T = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-141.816832, -75.339219], [-142.062846, -75.350338], [-141.948625,
        -75.438987], [-141.720183, -75.531503], [-141.5181, -75.522719], [-141.33359,
        -75.551256], [-141.096362, -75.507331], [-140.639478, -75.507331], [-140.336353,
        -75.420191], [-140.582367, -75.451136], [-140.946996, -75.490826], [-141.294052,
        -75.50073], [-141.298445, -75.451136], [-140.990927, -75.416874], [-140.617512,
        -75.400268], [-140.520864, -75.368111], [-140.358319, -75.296897], [-140.244098,
        -75.219734], [-140.428608, -75.221978], [-140.586761, -75.291319], [-140.828382,
        -75.323641], [-141.021679, -75.381423], [-141.324803, -75.425721], [-141.307231,
        -75.374771], [-141.386307, -75.334771], [-141.816832, -75.339219]]
)
###
OW8aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-137.126919, -74.797178], [-137.24114, -74.804092], [-137.236747, -74.839761],
         [-137.166457, -74.878786], [-137.017091, -74.886805], [-136.933622, -74.92114],
        [-136.929229, -74.97478], [-136.929229, -75.005512], [-136.819401, -74.959965],
        [-136.84576, -74.93143], [-136.880905, -74.893679], [-137.008305, -74.8501],
        [-137.021484, -74.821361], [-137.126919, -74.797178]]
)
OW8bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-137.796893, -74.915283], [-137.854003, -74.97122], [-137.750765, -74.986027],
         [-137.671689, -74.985457], [-137.660706, -74.963242], [-137.430068, -74.926143],
        [-137.430068, -74.90613], [-137.473999, -74.900981], [-137.649723, -74.93928],
        [-137.700244, -74.936996], [-137.748568, -74.891246], [-137.796893, -74.915283]]
)
OW8T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([OW8aT,OW8bT])
)
DW8aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-136.12529, -74.018634], [-135.760661, -74.137984], [-136.068179, -74.209892],
         [-136.375697, -74.232596], [-136.402056, -74.190747], [-136.098931, -74.152388],
        [-136.138469, -74.085045], [-136.287835, -74.082638], [-136.50749, -74.109127],
        [-136.494311, -74.159588], [-136.560208, -74.214674], [-136.665642, -74.175172],
        [-136.617318, -74.087457], [-136.301014, -74.023477], [-136.296621, -73.977446],
        [-136.12529, -74.018634]]
)
```

```
DW8bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-142.376688, -74.372], [-142.605129, -74.412199], [-142.504088, -74.438159],
        [-142.398653, -74.419283], [-141.932983, -74.499349], [-141.20812, -74.618688],
        [-140.94014, -74.777573], [-140.571119, -74.760255], [-140.720484, -74.66408],
        [-140.81274, -74.395659], [-140.02198, -74.411022], [-140.030766, -74.582511],
        [-139.529951, -74.59069], [-139.674924, -74.319827], [-140.531581, -74.248407],
        [-140.514008, -74.214966], [-139.081854, -74.123862], [-139.248792, -74.012884],
        [-139.6266, -74.017724], [-140.02198, -73.998351], [-140.342677, -74.061227],
        [-140.34707, -74.129871], [-141.142223, -74.212578], [-140.913781, -74.313887],
        [-141.13783, -74.453476], [-141.511245, -74.513433], [-142.19657, -74.380287],
        [-142.376688, -74.372]]
)
DW8T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([DW8aT,DW8bT])
)
FI8aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-140.789221, -74.943987], [-141.013269, -74.967372], [-140.685982,
        -75.080931], [-140.723324, -75.123299], [-140.54101, -75.164987], [-140.461934,
        -75.145284], [-140.514651, -75.086021], [-140.380661, -75.069613], [-140.161005,
        -75.026527], [-139.893026, -74.999823], [-139.759036, -74.93771], [-140.037998,
        -74.961672], [-140.573958, -75.030502], [-140.699162, -74.976488], [-140.789221,
        -74.943987]]
)
FI8bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-135.912696, -74.692968], [-136.033506, -74.679619], [-136.02472, -74.659869],
         [-135.939054, -74.636018], [-135.94784, -74.625536], [-136.090617, -74.599884],
        [-135.941251, -74.605134], [-135.822637, -74.628449], [-135.701826, -74.661032],
        [-135.873157, -74.649403], [-135.923678, -74.657544], [-135.912696, -74.692968]]
)
FI8cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-137.079067, -74.535568], [-137.197681, -74.540253], [-137.085656,
        -74.563664], [-136.947273, -74.556643], [-136.962649, -74.593463], [-137.07687,
        -74.614464], [-137.087853, -74.626119], [-136.929701, -74.637766], [-136.94947,
        -74.661611], [-136.881377, -74.646495], [-136.920915, -74.616212], [-136.931898,
        -74.581199], [-136.894556, -74.561325], [-136.989008, -74.542009], [-137.079067,
        -74.535568]]
)
FI8dT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-136.586566, -73.957118], [-136.709574, -73.943753], [-136.797436,
        -73.980176], [-136.819401, -74.03224], [-136.709574, -74.0564], [-136.617318,
        -74.03224], [-136.441594, -73.994725], [-136.454773, -73.952261], [-136.503097,
        -73.90481], [-136.604139, -73.919426], [-136.533849, -73.940108], [-136.586566,
        -73.957118]]
)
FI8eT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-140.683167, -74.854546], [-140.733688, -74.937565], [-140.601894,
        -74.928428], [-140.417384, -74.874055], [-140.252642, -74.874055], [-139.940731,
        -74.951265], [-139.725469, -74.886089], [-139.725469, -74.864304], [-139.679341,
        -74.83904], [-139.758417, -74.814885], [-139.863852, -74.769928], [-139.934142,
        -74.768195], [-139.868245, -74.806249], [-139.918766, -74.809128], [-140.03079,
        -74.756067], [-140.085704, -74.75838], [-140.092294, -74.794728], [-140.138421,
        -74.794728], [-140.180156, -74.776852], [-140.169173, -74.753178], [-140.085704,
        -74.734678], [-140.120849, -74.705145], [-140.208711, -74.688328], [-140.142814,
        -74.727154], [-140.215301, -74.757802], [-140.320735, -74.796455], [-140.476691,
        -74.807976], [-140.669988, -74.838465], [-140.683167, -74.854546]]
)
FI8T = ee.Geometry.MultiLineString(
```

```
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([FI8aT,FI8bT,FI8cT,FI8dT,FI8eT])
)
RI8aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-139.545366, -74.210786], [-139.62554, -74.211682], [-139.589297, -74.214372],
         [-139.557447, -74.219153], [-139.540973, -74.219153], [-139.511319, -74.222739],
        [-139.473978, -74.225427], [-139.457504, -74.218257], [-139.492649, -74.221245],
        [-139.521204, -74.211981], [-139.545366, -74.210786]]
)
RI8bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-139.816641, -74.222215], [-139.944041, -74.219228], [-139.95722, -74.224008],
         [-139.920977, -74.223709], [-139.879243, -74.224905], [-139.833115, -74.224606],
        [-139.816641, -74.222215]]
)
RI8cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-135.379885, -74.574626], [-135.461158, -74.59682], [-135.048205, -74.616067],
         [-134.810977, -74.648676], [-134.775832, -74.873336], [-134.424383, -74.899689],
        [-134.01802, -74.959676], [-133.68634, -74.895108], [-134.253052, -74.694563],
        [-134.477101, -74.598567], [-134.685773, -74.628301], [-135.028436, -74.587476],
        [-135.164623, -74.584556], [-135.379885, -74.574626]]
)
RI8dT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-136.776895, -74.860572], [-136.803253, -74.775406], [-136.636315,
        -74.775406], [-136.464984, -74.765596], [-136.352959, -74.738435], [-136.260704,
        -74.709489], [-136.122321, -74.711806], [-136.388104, -74.777714], [-136.776895,
        -74.860572]]
)
RI8T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([RI8aT,RI8bT,RI8cT,RI8dT])
)
###
OW9aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-127.502386, -73.385363], [-127.462848, -73.373737], [-127.447473,
        -73.360213], [-127.450767, -73.352033], [-127.39146, -73.345106], [-127.412328,
        -73.340067], [-127.450767, -73.336602], [-127.490305, -73.327464], [-127.528745,
        -73.3303], [-127.496895, -73.339438], [-127.459554, -73.344162], [-127.487011,
        -73.360843], [-127.517762, -73.368391], [-127.535335, -73.376565], [-127.559497,
        -73.383164], [-127.539728, -73.388504], [-127.502386, -73.385363]]
)
OW9bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-127.585372, -74.054969], [-127.635893, -74.054063], [-127.638089,
        -74.041382], [-127.657858, -74.040475], [-127.653465, -74.052856], [-127.661153,
        -74.058288], [-127.639188, -74.062211], [-127.616124, -74.0604], [-127.583175,
        -74.061305], [-127.585372, -74.054969]]
)
OW9cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-128.82776, -74.100929], [-128.868397, -74.099726], [-128.900247, -74.084971],
         [-128.920016, -74.088585], [-128.894755, -74.094006], [-128.884871, -74.104842],
        [-128.82776, -74.100929]]
)
OW9T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([OW9aT,OW9bT,OW9cT])
```

```
)
DW9aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-125.158348, -73.26597], [-125.116613, -73.279252], [-124.991409, -73.26154],
        [-124.90794, -73.241914], [-124.945282, -73.220358], [-124.987016, -73.227334],
        [-124.995803, -73.240011], [-125.052913, -73.24698], [-125.158348, -73.26597]]
)
DW9bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-123.192613, -73.705801], [-123.192613, -73.692235], [-123.170647,
        -73.680509], [-123.091571, -73.673715], [-123.192613, -73.666919], [-123.236544,
        -73.68915], [-123.192613, -73.705801]]
)
DW9cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-126.468774, -73.248166], [-126.43912, -73.258929], [-126.446808, -73.239297],
         [-126.423744, -73.221229], [-126.422646, -73.197109], [-126.429236, -73.182493],
        [-126.458889, -73.168502], [-126.497329, -73.162138], [-126.54785, -73.162457],
        [-126.559931, -73.151634], [-126.585191, -73.139529], [-126.603862, -73.133793],
        [-126.646695, -73.133793], [-126.660972, -73.142396], [-126.629122, -73.156091],
        [-126.606059, -73.164048], [-126.615943, -73.188531], [-126.631319, -73.201554],
        [-126.592879, -73.208539], [-126.576405, -73.229474], [-126.576405, -73.251332],
        [-126.525884, -73.26241], [-126.500624, -73.269052], [-126.505017, -73.250065],
        [-126.539064, -73.238979], [-126.526983, -73.223766], [-126.514901, -73.202824],
        [-126.536867, -73.192344], [-126.551145, -73.175181], [-126.532474, -73.168183],
        [-126.490739, -73.172001], [-126.449005, -73.18599], [-126.443513, -73.213615],
        [-126.453398, -73.231374], [-126.474265, -73.238347], [-126.468774, -73.248166]]
)
DW9dT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-127.045281, -72.891536], [-127.080426, -72.837159], [-127.286902,
        -72.905752], [-127.387943, -72.958636], [-127.554882, -72.998515], [-127.502164,
        -73.109946], [-127.65153, -73.13164], [-127.704247, -73.171127], [-127.616385,
        -73.228285], [-127.405516, -73.297889], [-127.25615, -73.221943], [-127.225398,
        -73.130364], [-126.926667, -73.075443], [-127.023315, -73.012639], [-127.177074,
        -73.083112], [-127.335226, -73.080558], [-127.339619, -73.040865], [-127.032101,
        -72.963785], [-127.045281, -72.891536]]
)
DW9T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([DW9aT,DW9bT,DW9cT,DW9dT])
)
FI9aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-127.493767, -73.341366], [-127.483882, -73.336484], [-127.421281,
        -73.337429], [-127.391627, -73.335066], [-127.387234, -73.339477], [-127.41524,
        -73.345303], [-127.425125, -73.353016], [-127.445443, -73.356634], [-127.44709,
        -73.360725], [-127.453131, -73.368273], [-127.483333, -73.376604], [-127.484981,
        -73.381632], [-127.486628, -73.387915], [-127.508594, -73.392468], [-127.507495,
        -73.385873], [-127.502553, -73.380061], [-127.488276, -73.372675], [-127.506397,
        -73.365757], [-127.489923, -73.360567], [-127.463015, -73.359309], [-127.448738,
        -73.348137], [-127.465761, -73.346877], [-127.42787, -73.342469], [-127.438853,
        -73.339791], [-127.493767, -73.341366]]
)
FI9bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-125.864952, -73.246504], [-125.967091, -73.249354], [-125.946224,
        -73.283832], [-125.931947, -73.307516], [-125.949519, -73.318241], [-125.978074,
        -73.328331], [-125.979173, -73.340933], [-125.965993, -73.351953], [-125.92316,
        -73.362651], [-125.881426, -73.358562], [-125.839691, -73.370198], [-125.806743,
        -73.3768], [-125.747436, -73.389996], [-125.694719, -73.412278], [-125.587087,
        -73.410397], [-125.607955, -73.391252], [-125.678244, -73.373657], [-125.752927,
        -73.339674], [-125.811136, -73.31351], [-125.806743, -73.301519], [-125.702406,
        -73.318557], [-125.699112, -73.306884], [-125.801251, -73.278143], [-125.788072,
```

```
        -73.270871], [-125.687031, -73.282884], [-125.685932, -73.268341], [-125.684834,
        -73.254735], [-125.615642, -73.249354], [-125.565122, -73.232247], [-125.803448,
        -73.228125], [-125.864952, -73.246504]]
)
FI9cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-126.292246, -73.197983], [-126.353749, -73.253152], [-126.287853,
        -73.253786], [-126.243921, -73.227808], [-126.164845, -73.182734], [-126.120914,
        -73.152191], [-126.129701, -73.12032], [-126.096752, -73.103723], [-125.890276,
        -73.166197], [-125.776055, -73.212585], [-125.571775, -73.213855], [-125.582758,
        -73.176375], [-125.305992, -73.067286], [-124.961133, -73.046153], [-124.961133,
        -72.969096], [-125.255471, -72.933023], [-125.422409, -73.016015], [-125.729927,
        -73.114577], [-125.824379, -73.133075], [-125.86831, -73.07816], [-125.633279,
        -73.030769], [-125.426803, -72.95944], [-125.505879, -72.90527], [-125.795824,
        -72.92399], [-125.738714, -73.010877], [-125.914438, -73.017939], [-126.221956,
        -72.927861], [-126.463577, -72.936247], [-126.371322, -73.027563], [-126.112128,
        -73.067286], [-126.175828, -73.092225], [-126.331784, -73.07944], [-126.283459,
        -73.129249], [-126.27028, -73.151553], [-126.342767, -73.149006], [-126.439415,
        -73.112664], [-126.522884, -73.101169], [-126.536063, -73.130525], [-126.496525,
        -73.145183], [-126.40427, -73.160469], [-126.309818, -73.16556], [-126.292246,
        -73.197983]]
)
FI9dT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-128.418073, -73.318952], [-128.433448, -73.339753], [-128.405442,
        -73.360843], [-128.376887, -73.365876], [-128.367003, -73.360686], [-128.335702,
        -73.358798], [-128.331309, -73.34306], [-128.308245, -73.332191], [-128.286828,
        -73.333452], [-128.272551, -73.343532], [-128.280788, -73.322735], [-128.31154,
        -73.319582], [-128.338447, -73.313906], [-128.38128, -73.320214], [-128.414778,
        -73.313906], [-128.418073, -73.318952]]
)
FI9eT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-123.719369, -73.440797], [-124.480924, -73.450344], [-124.419421,
        -73.607339], [-124.149245, -73.636457], [-124.184389, -73.769025], [-124.153638,
        -73.832778], [-123.885658, -73.829106], [-123.523226, -73.817475], [-123.343108,
        -73.792348], [-123.257443, -73.721669], [-123.345305, -73.643266], [-123.496867,
        -73.650691], [-123.470509, -73.717356], [-123.604499, -73.71859], [-123.611088,
        -73.649452], [-123.890051, -73.580033], [-123.874675, -73.565739], [-123.362877,
        -73.610439], [-123.325536, -73.556412], [-123.626464, -73.547702], [-123.617678,
        -73.531514], [-123.393629, -73.516559], [-123.595712, -73.434689], [-123.719369,
        -73.440797]]
)
FI9T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([FI9aT,FI9bT,FI9cT,FI9dT,FI9eT])
)
RI9aT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-127.109144, -73.78468], [-127.267296, -74.002899], [-127.592386, -74.17877],
        [-127.913083, -74.292193], [-127.170648, -74.457898], [-125.307968, -74.143989],
        [-126.753302, -73.771175], [-127.109144, -73.78468]]
)
RI9bT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-126.742017, -73.056483], [-126.799127, -73.041428], [-126.811209,
        -73.040467], [-126.824937, -73.046554], [-126.74037, -73.064485], [-126.742017,
        -73.056483]]
)
RI9cT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-126.139225, -73.299862], [-126.23807, -73.291021], [-126.386337, -73.291968],
        [-126.452234, -73.300178], [-126.457725, -73.347153], [-126.119456, -73.351875],
        [-126.029397, -73.370749], [-125.72737, -73.574827], [-125.439622, -73.640248],
```

```
        [-125.428639, -73.598731], [-125.624132, -73.546223], [-125.699913, -73.509152],
        [-125.618641, -73.443222], [-125.664768, -73.433515], [-125.743844, -73.429441],
        [-125.776793, -73.414081], [-125.819626, -73.405611], [-125.853672, -73.378293],
        [-125.900898, -73.384577], [-125.967893, -73.375463], [-126.048067, -73.348412],
        [-126.076623, -73.326045], [-126.123849, -73.3128], [-126.112866, -73.29986],
        [-126.139225, -73.299862]]
)
RI9dT = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-128.481373, -73.206873], [-128.49016, -73.20973], [-128.446229, -73.213855],
        [-128.40724, -73.217186], [-128.416026, -73.213062], [-128.481373, -73.206873]]
)
RI9T = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([RI9aT,RI9bT,RI9cT,RI9dT])
)

## (3.2) Training Feature Collection
TrainingRings = ee.FeatureCollection([
    ee.Feature(OW1T,{
        'classes': 1,
        'subclasses': 1,
        'name': 'Open Water',
        'type': 'Water',
        'image': 'T1',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(0)).get('system:
            time_start'),
    }),
    ee.Feature(DW1T,{
        'classes': 1,
        'subclasses': 2,
        'name': 'Rough Water',
        'type': 'Water',
        'image': 'T1',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(0)).get('system:
            time_start'),
    }),
    ee.Feature(FI1T,{
        'classes': 2,
        'subclasses': 3,
        'name': 'Flat Sea Ice',
        'type': 'Ice',
        'image': 'T1',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(0)).get('system:
            time_start')
    }),
    ee.Feature(RI1T,{
        'classes': 2,
        'subclasses': 4,
        'name': 'Floating Land Ice',
        'type': 'Ice',
        'image': 'T1',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(0)).get('system:
            time_start')
    }),

    ee.Feature(OW2T,{
        'classes': 1,
        'subclasses': 1,
        'name': 'Open Water',
        'type': 'Water',
        'image': 'T2',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(1)).get('system:
            time_start'),
    }),
    ee.Feature(DW2T,{
        'classes': 1,
        'subclasses': 2,
        'name': 'Rough Water',
```

```
        'type': 'Water',
        'image': 'T2',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(1)).get('system:time_start'),
    }),
    ee.Feature(FI2T,{
        'classes': 2,
        'subclasses': 3,
        'name': 'Flat Sea Ice',
        'type': 'Ice',
        'image': 'T2',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(1)).get('system:time_start')
    }),
    ee.Feature(RI2T,{
        'classes': 2,
        'subclasses': 4,
        'name': 'Floating Land Ice',
        'type': 'Ice',
        'image': 'T2',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(1)).get('system:time_start')
    }),

    ee.Feature(OW3T,{
        'classes': 1,
        'subclasses': 1,
        'name': 'Open Water',
        'type': 'Water',
        'image': 'T3',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(2)).get('system:time_start'),
    }),
    ee.Feature(DW3T,{
        'classes': 1,
        'subclasses': 2,
        'name': 'Rough Water',
        'type': 'Water',
        'image': 'T3',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(2)).get('system:time_start'),
    }),
    ee.Feature(FI3T,{
        'classes': 2,
        'subclasses': 3,
        'name': 'Flat Sea Ice',
        'type': 'Ice',
        'image': 'T3',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(2)).get('system:time_start')
    }),
    ee.Feature(RI3T,{
        'classes': 2,
        'subclasses': 4,
        'name': 'Floating Land Ice',
        'type': 'Ice',
        'image': 'T3',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(2)).get('system:time_start')
    }),

    ee.Feature(OW4T,{
        'classes': 1,
        'subclasses': 1,
        'name': 'Open Water',
        'type': 'Water',
        'image': 'T4',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(3)).get('system:time_start'),
    }),
    ee.Feature(DW4T,{
```

```
        'classes': 1,
        'subclasses': 2,
        'name': 'Rough Water',
        'type': 'Water',
        'image': 'T4',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(3)).get('system:time_start'),
    }),
    ee.Feature(FI4T,{
        'classes': 2,
        'subclasses': 3,
        'name': 'Flat Sea Ice',
        'type': 'Ice',
        'image': 'T4',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(3)).get('system:time_start')
    }),
    ee.Feature(RI4T,{
        'classes': 2,
        'subclasses': 4,
        'name': 'Floating Land Ice',
        'type': 'Ice',
        'image': 'T4',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(3)).get('system:time_start')
    }),

    ee.Feature(OW5T,{
        'classes': 1,
        'subclasses': 1,
        'name': 'Open Water',
        'type': 'Water',
        'image': 'T5',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(4)).get('system:time_start'),
    }),
    ee.Feature(DW5T,{
        'classes': 1,
        'subclasses': 2,
        'name': 'Rough Water',
        'type': 'Water',
        'image': 'T5',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(4)).get('system:time_start'),
    }),
    ee.Feature(FI5T,{
        'classes': 2,
        'subclasses': 3,
        'name': 'Flat Sea Ice',
        'type': 'Ice',
        'image': 'T5',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(4)).get('system:time_start')
    }),
    ee.Feature(RI5T,{
        'classes': 2,
        'subclasses': 4,
        'name': 'Floating Land Ice',
        'type': 'Ice',
        'image': 'T5',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(4)).get('system:time_start')
    }),

    ee.Feature(OW6T,{
        'classes': 1,
        'subclasses': 1,
        'name': 'Open Water',
        'type': 'Water',
        'image': 'T6',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(5)).get('system:
```

```
            time_start'),
    }),
    ee.Feature(DW6T,{
        'classes': 1,
        'subclasses': 2,
        'name': 'Rough Water',
        'type': 'Water',
        'image': 'T6',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(5)).get('system:
            time_start'),
    }),
    ee.Feature(FI6T,{
        'classes': 2,
        'subclasses': 3,
        'name': 'Flat Sea Ice',
        'type': 'Ice',
        'image': 'T6',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(5)).get('system:
            time_start')
    }),
    ee.Feature(RI6T,{
        'classes': 2,
        'subclasses': 4,
        'name': 'Floating Land Ice',
        'type': 'Ice',
        'image': 'T6',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(5)).get('system:
            time_start')
    }),

    ee.Feature(OW7T,{
        'classes': 1,
        'subclasses': 1,
        'name': 'Open Water',
        'type': 'Water',
        'image': 'T7',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(6)).get('system:
            time_start'),
    }),
    ee.Feature(DW7T,{
        'classes': 1,
        'subclasses': 2,
        'name': 'Rough Water',
        'type': 'Water',
        'image': 'T7',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(6)).get('system:
            time_start'),
    }),
    ee.Feature(FI7T,{
        'classes': 2,
        'subclasses': 3,
        'name': 'Flat Sea Ice',
        'type': 'Ice',
        'image': 'T7',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(6)).get('system:
            time_start')
    }),
    ee.Feature(RI7T,{
        'classes': 2,
        'subclasses': 4,
        'name': 'FLoating Land Ice',
        'type': 'Ice',
        'image': 'T7',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(6)).get('system:
            time_start')
    }),

    ee.Feature(OW8T,{
        'classes': 1,
        'subclasses': 1,
        'name': 'Open Water',
```

```
        'type': 'Water',
        'image': 'T8',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(7)).get('system:
            time_start'),
    }),
    ee.Feature(DW8T,{
        'classes': 1,
        'subclasses': 2,
        'name': 'Rough Water',
        'type': 'Water',
        'image': 'T8',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(7)).get('system:
            time_start'),
    }),
    ee.Feature(FI8T,{
        'classes': 2,
        'subclasses': 3,
        'name': 'Flat Sea Ice',
        'type': 'Ice',
        'image': 'T8',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(7)).get('system:
            time_start')
    }),
    ee.Feature(RI8T,{
        'classes': 2,
        'subclasses': 4,
        'name': 'Floating Land Ice',
        'type': 'Ice',
        'image': 'T8',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(7)).get('system:
            time_start')
    }),

    ee.Feature(OW9T,{
        'classes': 1,
        'subclasses': 1,
        'name': 'Open Water',
        'type': 'Water',
        'image': 'T9',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(8)).get('system:
            time_start'),
    }),
    ee.Feature(DW9T,{
        'classes': 1,
        'subclasses': 2,
        'name': 'Rough Water',
        'type': 'Water',
        'image': 'T9',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(8)).get('system:
            time_start'),
    }),
    ee.Feature(FI9T,{
        'classes': 2,
        'subclasses': 3,
        'name': 'Flat Sea Ice',
        'type': 'Ice',
        'image': 'T9',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(8)).get('system:
            time_start')
    }),
    ee.Feature(RI9T,{
        'classes': 2,
        'subclasses': 4,
        'name': 'Floating Land Ice',
        'type': 'Ice',
        'image': 'T9',
        'utc': ee.Image(TrainingData.toList(TrainingData.size()).get(8)).get('system:
            time_start')
    })
])
```

```
TrainingPolys = TrainingRings.map(lambda feature: ee.Feature(ee.Geometry.MultiPolygon(feature
    .geometry().coordinates())).copyProperties(feature))

## (3.3) Validation Polygons
OW1aV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-106.447642, -74.989868], [-106.446544, -74.984176], [-106.41689, -74.981614],
         [-106.396023, -74.987307], [-106.380647, -74.992713], [-106.419087, -74.992429],
        [-106.447642, -74.989868]]
)
OW1bV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-105.9097, -74.317455], [-106.160107, -74.376739], [-106.647743, -74.382655],
        [-106.296294, -74.471139], [-105.993169, -74.507571], [-105.672472, -74.536889],
        [-105.351775, -74.540404], [-105.659293, -74.469964], [-105.567037, -74.451126],
        [-105.334202, -74.45937], [-105.044257, -74.42873], [-105.294664, -74.375557],
        [-105.76912, -74.317455], [-105.9097, -74.317455]]
)
OW1V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([OW1aV,OW1bV])
)
DW1aV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-107.619805, -74.028236], [-107.606626, -74.023701], [-107.580267,
        -74.026725], [-107.55281, -74.02491], [-107.497896, -74.022491], [-107.493503,
        -74.032165], [-107.54622, -74.033979], [-107.561596, -74.04274], [-107.555006,
        -74.051195], [-107.51437, -74.048779], [-107.528648, -74.054515], [-107.593446,
        -74.050289], [-107.614314, -74.038813], [-107.619805, -74.028236]]
)
DW1bV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-105.299057, -74.13528], [-105.44403, -74.191647], [-105.597789, -74.226326],
        [-105.408885, -74.269276], [-105.189229, -74.266891], [-104.965181, -74.271658],
        [-104.833387, -74.272848], [-104.947608, -74.211985], [-105.039864, -74.170083],
        [-105.299057, -74.13528]]
)
DW1cV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-107.237293, -74.300521], [-107.3603, -74.299331], [-107.281224, -74.325471],
        [-107.193361, -74.353943], [-107.101106, -74.397734], [-106.973706, -74.415453],
        [-107.09232, -74.37171], [-107.237293, -74.300521]]
)
DW1V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([DW1aV,DW1bV,DW1cV])
)
FI1aV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-107.042918, -74.449355], [-107.262573, -74.511674], [-107.223035, -74.57258],
         [-107.38558, -74.651874], [-107.350435, -74.714558], [-106.766151, -74.64373],
        [-106.643144, -74.685568], [-107.00338, -74.759677], [-106.832048, -74.825383],
        [-106.695862, -74.828834], [-106.265337, -74.854115], [-106.252157, -74.790835],
        [-106.094006, -74.712242], [-106.34002, -74.640238], [-106.322447, -74.510499],
        [-106.845228, -74.488186], [-107.042918, -74.449355]]
)
FI1bV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-107.06049, -73.932808], [-106.656324, -74.011676], [-107.104422, -74.11785],
        [-106.700255, -74.287728], [-106.502565, -74.230502], [-106.14233, -74.218555],
        [-105.940247, -74.237668], [-105.812847, -74.223337], [-105.87435, -74.161087],
        [-106.278516, -74.129871], [-106.26973, -74.103413], [-106.032502, -74.101006],
```

```
        [-106.023716, -74.063643], [-105.896316, -74.040702], [-106.036895, -74.017727],
        [-106.217013, -74.03224], [-106.401524, -73.986239], [-106.34002, -73.965614],
        [-106.884766, -73.90603], [-107.06049, -73.932808]]
)
FI1cV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-109.881836, -74.558843], [-109.996057, -74.564106], [-109.86646, -74.628889],
         [-109.767615, -74.623646], [-109.741257, -74.599157], [-109.655591, -74.579889],
        [-109.479867, -74.565276], [-109.550156, -74.528391], [-109.651198, -74.499057],
        [-109.620446, -74.53308], [-109.73906, -74.534838], [-109.800564, -74.544795],
        [-109.701719, -74.556502], [-109.826922, -74.596239], [-109.859871, -74.577552],
        [-109.837905, -74.560597], [-109.881836, -74.558843]]
)
FI1V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([FI1aV,FI1bV,FI1cV])
)
RI1aV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-108.989101, -74.225131], [-107.75903, -74.025591], [-107.785388, -74.215567],
         [-108.075334, -74.287132], [-108.391638, -74.303789], [-108.54979, -74.263312],
        [-108.866094, -74.268083], [-108.989101, -74.225131]]
)
RI1bV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-107.763388, -75.13204], [-108.092871, -75.192808], [-107.521767, -75.197301],
         [-107.218642, -75.237665], [-106.814476, -75.271221], [-106.546496, -75.275691],
        [-106.401523, -75.224222], [-106.550889, -75.170329], [-106.621179, -75.115116],
        [-106.53771, -74.997263], [-106.788117, -74.972213], [-106.748579, -75.077818],
        [-106.774938, -75.164704], [-106.897945, -75.215253], [-107.218642, -75.204034],
        [-107.253787, -75.171455], [-107.376794, -75.133168], [-107.763388, -75.13204]]
)
RI1cV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-107.391052, -74.986169], [-107.536024, -75.0021], [-107.650245, -75.025962],
        [-107.768859, -75.074709], [-107.663424, -75.11766], [-107.386659, -75.112016],
        [-107.311975, -75.062256], [-107.224113, -75.03391], [-107.263651, -74.982755],
        [-107.391052, -74.986169]]
)
RI1V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([RI1aV,RI1bV,RI1cV])
)
####
OW2aV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-102.891684, -73.380105], [-102.913649, -73.406478], [-102.891684,
        -73.449097], [-102.658849, -73.44534], [-102.610525, -73.486611], [-102.557807,
        -73.415261], [-102.755497, -73.387643], [-102.891684, -73.380105]]
)
OW2bV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-103.597267, -73.218775], [-103.60825, -73.239697], [-103.559926, -73.254897],
         [-103.522584, -73.244764], [-103.597267, -73.218775]]
)
OW2V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([OW2aV,OW2bV])
)
DW2aV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
```

```
    coords = [[-101.989228, -75.052905], [-102.033159, -75.071596], [-101.894776,
        -75.090263], [-101.758589, -75.063103], [-101.642172, -75.064801], [-101.374192,
        -75.028516], [-101.328064, -75.019999], [-101.130374, -75.005225], [-100.893146,
        -74.960818], [-100.710832, -74.933426], [-100.710832, -74.917997], [-100.748174,
        -74.916854], [-100.609791, -74.867603], [-100.789908, -74.837173], [-100.829446,
        -74.860145], [-100.787712, -74.891676], [-100.983205, -74.928857], [-101.035922,
        -74.916283], [-101.090836, -74.924285], [-101.038119, -74.937424], [-101.128178,
        -74.957397], [-101.169912, -75.000676], [-101.41373, -75.018295], [-101.462054,
        -75.029083], [-101.730034, -75.046102], [-101.892579, -75.058004], [-101.989228,
        -75.052905]]
)
DW2bV = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-104.336548, -73.920186], [-104.315681, -73.925358], [-104.244293,
        -73.912273], [-104.204755, -73.88607], [-104.166315, -73.87753], [-104.111401,
        -73.874479], [-104.096026, -73.868375], [-104.117991, -73.862574], [-104.175102,
        -73.871732], [-104.21464, -73.876614], [-104.256374, -73.900395], [-104.311288,
        -73.905269], [-104.346433, -73.911968], [-104.336548, -73.920186]]
)
DW2cV = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-101.440654, -73.603305], [-101.495568, -73.605786], [-101.486781,
        -73.623142], [-101.440654, -73.616325], [-101.427475, -73.650998], [-101.471406,
        -73.649142], [-101.477995, -73.660891], [-101.374757, -73.655945], [-101.390133,
        -73.626859], [-101.440654, -73.603305]]
)
DW2dV = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-103.827473, -73.478484], [-103.572673, -73.523413], [-103.348624,
        -73.651305], [-103.480417, -73.66367], [-103.673714, -73.610436], [-103.603425,
        -73.56698], [-103.682501, -73.555788], [-103.792328, -73.586859], [-103.76597,
        -73.614155], [-103.845046, -73.614155], [-103.875797, -73.565738], [-103.849439,
        -73.51718], [-103.910943, -73.487229], [-103.827473, -73.478484]]
)
DW2eV = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-104.304052, -72.80764], [-104.6555, -72.8647], [-104.541279, -72.904793],
        [-104.128327, -72.930611], [-103.891099, -73.075768], [-103.592367, -73.187982],
        [-103.592367, -72.943505], [-104.304052, -72.80764]]
)
DW2V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = ee.List([DW2aV,DW2bV,DW2cV,DW2dV,DW2eV])
)
FI2aV = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-101.83273, -73.338573], [-102.236896, -73.34361], [-102.245682, -73.406475],
        [-102.447765, -73.541481], [-101.973309, -73.588725], [-101.736081, -73.633365],
        [-101.62186, -73.576305], [-101.83273, -73.486605], [-101.83273, -73.338573]]
)
FI2bV = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-103.062801, -73.845006], [-103.712982, -73.896284], [-103.563615,
        -74.029825], [-103.001298, -74.066058], [-102.772855, -74.167083], [-102.597131,
        -74.265104], [-102.043599, -74.345937], [-101.982095, -74.234088], [-101.586715,
        -74.243635], [-101.50764, -74.128674], [-101.727295, -74.024989], [-102.149034,
        -73.995938], [-102.913435, -73.940108], [-103.062801, -73.845006]]
)
FI2cV = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-105.220969, -73.326285], [-105.225362, -73.385444], [-105.155072,
        -73.405535], [-105.317617, -73.440641], [-105.339583, -73.478174], [-105.3923,
        -73.531828], [-105.128713, -73.534318], [-105.119927, -73.572889], [-104.957382,
```

```
        -73.567915], [-104.764085, -73.594002], [-104.482926, -73.580342], [-104.491712,
        -73.544281], [-104.834375, -73.562943], [-104.829982, -73.520613], [-104.944203,
        -73.511882], [-104.878306, -73.481923], [-104.98374, -73.456916], [-104.917844,
        -73.438136], [-104.566395, -73.494413], [-104.368705, -73.485672], [-104.40385,
        -73.458168], [-104.500498, -73.451911], [-104.430208, -73.416828], [-104.58836,
        -73.382931], [-104.865126, -73.410556], [-104.843161, -73.391725], [-104.715761,
        -73.367841], [-104.79923, -73.348964], [-104.957382, -73.366585], [-105.220969,
        -73.326285]]
)
FI2dV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-104.002142, -73.351324], [-104.085611, -73.429207], [-103.793469,
        -73.445492], [-103.644104, -73.412279], [-104.002142, -73.351324]]
)
FI2eV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-103.247507, -73.408828], [-103.203576, -73.405691], [-103.196986,
        -73.393763], [-103.155251, -73.395647], [-103.183807, -73.418237], [-103.247507,
        -73.408828]]
)
FI2fV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-103.243355, -73.602609], [-103.262025, -73.622136], [-103.238962,
        -73.647828], [-103.213701, -73.648757], [-103.216996, -73.638237], [-103.141215,
        -73.621827], [-103.091792, -73.60478], [-103.092891, -73.595784], [-103.06763,
        -73.592992], [-102.931444, -73.589267], [-102.921559, -73.566594], [-103.051156,
        -73.548869], [-103.081908, -73.582437], [-103.143412, -73.592992], [-103.211505,
        -73.602919], [-103.243355, -73.602609]]
)
FI2gV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-102.84735, -74.759389], [-102.513474, -75.024541], [-102.333356, -74.994988],
         [-102.445381, -74.956255], [-102.335553, -74.912849], [-102.355322, -74.868749],
        [-102.269656, -74.865881], [-101.913814, -74.912278], [-101.639245, -74.925998],
        [-101.643638, -74.826248], [-101.819362, -74.822221], [-101.7315, -74.873337],
        [-102.166418, -74.854978], [-102.258674, -74.834297], [-102.194973, -74.795159],
        [-102.146649, -74.760544], [-102.84735, -74.759389]]
)
FI2V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([FI2aV,FI2bV,FI2cV,FI2dV,FI2eV,FI2fV,FI2gV])
)
RI2aV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-101.314342, -73.419021], [-101.454922, -73.499091], [-101.173763,
        -73.616021], [-101.182549, -73.714896], [-100.611444, -73.73706], [-99.882188,
        -73.727214], [-99.996409, -73.670473], [-101.314342, -73.419021]]
)
RI2bV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-105.012015, -72.950586], [-105.135022, -72.994981], [-105.108663,
        -73.064885], [-104.889008, -73.187976], [-104.71548, -73.203857], [-104.566114,
        -73.176533], [-104.678139, -73.110269], [-104.77918, -73.077681], [-104.761608,
        -73.047596], [-104.904384, -73.009113], [-105.012015, -72.950586]]
)
RI2cV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-102.276639, -75.192531], [-101.854901, -75.386696], [-100.730264,
        -74.961672], [-101.037782, -75.020849], [-101.521024, -75.077536], [-101.802183,
        -75.102414], [-102.276639, -75.192531]]
)
RI2dV = ee.Geometry.LineString(
    geodesic = False,
```

```
    proj = 'EPSG:4326',
    coords = [[-100.94074, -74.852968], [-101.063747, -74.858708], [-101.002244, -74.890816],
         [-100.986868, -74.913708], [-100.848485, -74.890816], [-100.866057, -74.866741],
        [-100.94074, -74.852968]]
)
RI2eV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-101.799593, -75.002808], [-101.966532, -75.005082], [-101.973121,
        -75.037452], [-101.874276, -75.043692], [-101.694159, -75.030075], [-101.593117,
        -75.029508], [-101.630458, -75.019288], [-101.751269, -75.013607], [-101.799593,
        -75.002808]]
)
RI2fV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-101.371265, -74.957253], [-101.509648, -74.965805], [-101.606296,
        -74.960104], [-101.665603, -74.964095], [-101.509648, -74.973782], [-101.362479,
        -74.988587], [-101.318548, -75.00167], [-101.268027, -74.993707], [-101.338316,
        -74.962384], [-101.371265, -74.957253]]
)
RI2gV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-101.481712, -74.880646], [-101.500383, -74.878353], [-101.516857,
        -74.880359], [-101.500143, -74.882652], [-101.481712, -74.880646]]
)
RI2V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([RI2aV,RI2bV,RI2cV,RI2dV,RI2eV,RI2fV,RI2gV])
)
###
OW3aV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-111.820533, -74.101305], [-111.923771, -74.116346], [-111.840302,
        -74.145186], [-111.785388, -74.149988], [-111.805157, -74.170379], [-111.877644,
        -74.198524], [-111.80955, -74.183559], [-111.721688, -74.178169], [-111.679954,
        -74.154789], [-111.717295, -74.148187], [-111.699723, -74.131975], [-111.631629,
        -74.122961], [-111.517408, -74.136781], [-111.475674, -74.134377], [-111.517408,
        -74.104917], [-111.589895, -74.105518], [-111.664578, -74.113339], [-111.704116,
        -74.098898], [-111.660185, -74.088661], [-111.684347, -74.075402], [-111.686543,
        -74.047648], [-111.695329, -74.03073], [-111.655792, -74.003504], [-111.701919,
        -73.987147], [-111.737064, -74.004715], [-111.706312, -74.019845], [-111.717295,
        -74.084443], [-111.750243, -74.125966], [-111.820533, -74.101305]]
)
OW3bV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-112.048975, -73.752127], [-111.974292, -73.774245], [-111.930361,
        -73.754587], [-111.934754, -73.726287], [-111.978685, -73.707187], [-112.024813,
        -73.731827], [-112.048975, -73.752127]]
)
OW3cV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-113.672752, -73.975854], [-113.797956, -73.996009], [-113.775441,
        -74.007969], [-113.627173, -73.993132], [-113.552491, -73.998281], [-113.442114,
        -73.968574], [-113.514051, -73.948234], [-113.534369, -73.952334], [-113.512953,
        -73.963568], [-113.475062, -73.964933], [-113.486594, -73.976916], [-113.583791,
        -73.985708], [-113.628272, -73.976461], [-113.672752, -73.975854]]
)
OW3V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([OW3aV,OW3bV,OW3cV])
)
DW3aV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
```

```
    coords = [[-108.904859, -74.336743], [-108.972952, -74.340895], [-108.964166,
        -74.360455], [-108.89168, -74.368745], [-108.860928, -74.385906], [-108.724741,
        -74.396546], [-108.904859, -74.336743]]
)
DW3bV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-111.097739, -73.541475], [-111.308608, -73.589965], [-111.708381,
        -73.682821], [-111.774278, -73.802156], [-111.717168, -73.916989], [-111.484333,
        -73.991086], [-111.21196, -73.980173], [-110.81658, -74.04674], [-110.856118,
        -73.916989], [-110.965946, -73.836446], [-111.01427, -73.732135], [-111.058201,
        -73.655634], [-110.952766, -73.557654], [-111.097739, -73.541475]]
)
DW3cV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-112.894364, -73.852952], [-113.335871, -73.930985], [-113.131592,
        -74.047346], [-113.36882, -74.098597], [-113.331478, -74.167682], [-113.188702,
        -74.178469], [-113.045926, -74.186255], [-112.925115, -74.177869], [-112.835057,
        -74.185056], [-112.753784, -74.198226], [-112.62858, -74.099801], [-112.920722,
        -74.064246], [-112.883381, -74.046138], [-112.439676, -74.029824], [-112.50118,
        -73.979569], [-112.786732, -73.97047], [-112.777946, -73.941929], [-112.712049,
        -73.817472], [-112.894364, -73.852952]]
)
DW3V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([DW3aV,DW3bV,DW3cV])
)
FI3aV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-109.02567, -74.585142], [-109.170642, -74.633544], [-109.045439, -74.624808],
         [-108.975149, -74.63005], [-108.858731, -74.616649], [-108.911449, -74.599737],
        [-109.02567, -74.585142]]
)
FI3bV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-109.924061, -74.547573], [-109.994351, -74.552256], [-109.948223,
        -74.600758], [-109.779088, -74.607174], [-109.748337, -74.577986], [-109.836199,
        -74.572141], [-109.812037, -74.537031], [-109.684636, -74.53293], [-109.673654,
        -74.563371], [-109.405674, -74.537031], [-109.35735, -74.520033], [-109.346367,
        -74.488921], [-109.247522, -74.487746], [-109.247522, -74.454215], [-109.322205,
        -74.434179], [-109.412263, -74.458925], [-109.394691, -74.497146], [-109.513305,
        -74.503018], [-109.596774, -74.49597], [-109.656081, -74.522378], [-109.838395,
        -74.52062], [-109.882326, -74.538204], [-109.924061, -74.547573]]
)
FI3cV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-109.470442, -74.169631], [-109.571484, -74.182213], [-109.549518,
        -74.193589], [-109.406742, -74.195383], [-109.395759, -74.185207], [-109.347435,
        -74.18341], [-109.426511, -74.169631], [-109.470442, -74.169631]]
)
FI3V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([FI3aV,FI3bV,FI3cV])
)
RI3aV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-112.196272, -74.245424], [-113.303337, -74.238262], [-113.443916, -74.58777],
         [-113.29455, -74.874766], [-112.213844, -74.941133], [-112.354425, -75.197016],
        [-111.229788, -75.277643], [-111.089208, -75.163295], [-109.402253, -75.199263],
        [-109.630695, -75.029931], [-109.059591, -74.863295], [-109.542833, -74.768916],
        [-109.331964, -74.676283], [-109.903068, -74.65071], [-110.087579, -74.831134],
        [-110.685042, -74.959386], [-111.64274, -74.952547], [-112.521362, -74.545669],
        [-112.196272, -74.245424]]
)
```

```
RI3bV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.322649, -73.806139], [-110.384152, -73.824512], [-110.29629, -73.864864],
         [-110.305077, -74.003503], [-110.454442, -74.041003], [-110.375366, -74.097693],
        [-110.204035, -74.074797], [-109.975593, -74.083238], [-109.8438, -74.166182],
        [-109.602179, -74.149388], [-109.567034, -74.09408], [-110.322649, -73.806139]]
)
RI3cV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.836461, -74.084444], [-110.863918, -74.093179], [-110.89467, -74.106422],
         [-110.871606, -74.105519], [-110.836461, -74.084444]]
)
RI3dV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-113.391858, -73.940258], [-113.37099, -73.947247], [-113.346828, -73.962126],
         [-113.350123, -73.971229], [-113.375383, -73.958787], [-113.389661, -73.948158],
        [-113.391858, -73.940258]]
)
RI3eV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-110.447421, -73.871123], [-110.531988, -73.861965], [-110.662683, -73.86746],
         [-110.624243, -73.880886], [-110.659388, -73.889425], [-110.604474, -73.904052],
        [-110.572624, -73.892778], [-110.515514, -73.893082], [-110.489155, -73.884545],
        [-110.447421, -73.871123]]
)
RI3V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([RI3aV,RI3bV,RI3cV,RI3dV,RI3eV])
)
###
OW4aV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-125.742381, -73.197666], [-125.883984, -73.193854], [-125.8291, -73.21671],
        [-125.764335, -73.207824], [-125.70945, -73.210998], [-125.68091, -73.209095],
        [-125.656761, -73.200523], [-125.631514, -73.19576], [-125.675422, -73.196078],
        [-125.699571, -73.201475], [-125.742381, -73.197666]]
)
OW4bV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-126.447153, -73.232724], [-126.484475, -73.234626], [-126.452641,
        -73.257112], [-126.448251, -73.251413], [-126.447153, -73.232724]]
)
OW4V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([OW4aV,OW4bV])
)
DW4aV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-123.893053, -73.430461], [-123.743687, -73.478644], [-123.589928,
        -73.473642], [-123.607501, -73.50237], [-123.42958, -73.536654], [-123.319752,
        -73.547238], [-123.251659, -73.520455], [-123.412007, -73.494258], [-123.447152,
        -73.506738], [-123.475707, -73.496129], [-123.475707, -73.481768], [-123.539407,
        -73.466768], [-123.644842, -73.463015], [-123.805191, -73.436726], [-123.893053,
        -73.430461]]
)
DW4bV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-124.237912, -73.341252], [-124.222536, -73.354475], [-124.15664, -73.367688],
         [-124.092939, -73.382146], [-124.068777, -73.404753], [-124.031436, -73.395337],
        [-124.044615, -73.378374], [-124.018257, -73.367688], [-124.064384, -73.361398],
        [-124.114905, -73.368317], [-124.145657, -73.352588], [-124.237912, -73.341252]]
```

```
)
DW4cV = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-123.930394, -73.599898], [-123.890856, -73.621597], [-123.829353,
        -73.641413], [-123.723918, -73.651313], [-123.666808, -73.642033], [-123.62068,
        -73.643269], [-123.658021, -73.630268], [-123.743687, -73.638938], [-123.840336,
        -73.627172], [-123.930394, -73.599898]]
)
DW4dV = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-123.322748, -73.70426], [-123.301331, -73.701177], [-123.283759, -73.695318],
         [-123.259048, -73.696243], [-123.27662, -73.701948], [-123.283759, -73.705955],
        [-123.322748, -73.70426]]
)
DW4eV = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-125.629844, -73.222259], [-125.761568, -73.26407], [-125.66058, -73.296939],
        [-125.537637, -73.307041], [-125.445431, -73.307673], [-125.430063, -73.27166],
        [-125.357615, -73.271027], [-125.342247, -73.289361], [-125.243454, -73.28999],
        [-125.190764, -73.259006], [-125.076604, -73.233036], [-125.012937, -73.21528],
        [-124.927317, -73.204489], [-125.091972, -73.159354], [-125.13588, -73.175894],
        [-125.102949, -73.20576], [-125.146857, -73.219088], [-125.232477, -73.200046],
        [-125.326879, -73.200681], [-125.390546, -73.257742], [-125.462994, -73.245078],
        [-125.425672, -73.219723], [-125.504706, -73.217818], [-125.629844, -73.222259]]
)
DW4fV = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-126.627175, -73.153546], [-126.55363, -73.167868], [-126.534969, -73.191711],
         [-126.498745, -73.182178], [-126.485572, -73.189805], [-126.51521, -73.200922],
        [-126.518503, -73.214569], [-126.627175, -73.153546]]
)
DW4V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = ee.List([DW4aV,DW4bV,DW4cV,DW4dV,DW4eV,DW4fV])
)
FI4aV = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-125.938297, -73.223207], [-126.015136, -73.222572], [-125.986596,
        -73.249191], [-125.907562, -73.275137], [-125.903171, -73.318711], [-125.95586,
        -73.341402], [-125.821941, -73.360918], [-125.694608, -73.409922], [-125.62216,
        -73.413059], [-125.648505, -73.38795], [-125.758275, -73.357142], [-125.821941,
        -73.309247], [-125.775838, -73.301672], [-125.685827, -73.312402], [-125.635333,
        -73.335102], [-125.608988, -73.309247], [-125.720953, -73.304199], [-125.76047,
        -73.28904], [-125.81316, -73.273871], [-125.848286, -73.251724], [-125.879022,
        -73.237155], [-125.938297, -73.223207]]
)
FI4bV = ee.Geometry.LineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = [[-124.358732, -73.343291], [-124.62218, -73.285248], [-124.481675, -73.407413],
         [-124.5651, -73.498774], [-124.270917, -73.630571], [-124.139193, -73.641713],
        [-124.12602, -73.811647], [-123.897699, -73.816551], [-123.340068, -73.821449],
        [-123.274206, -73.720737], [-123.476183, -73.72197], [-123.783538, -73.677566],
        [-123.932825, -73.636762], [-123.994297, -73.575989], [-123.884527, -73.569776],
        [-123.722067, -73.610742], [-123.520091, -73.582201], [-123.511309, -73.55236],
        [-123.638642, -73.533686], [-123.783538, -73.536179], [-123.954779, -73.473792],
        [-123.897699, -73.549875], [-123.985515, -73.55236], [-124.090894, -73.497525],
        [-124.17871, -73.393602], [-124.358732, -73.343291]]
)
FI4V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = ’EPSG:4326’,
    coords = ee.List([FI4aV,FI4bV])
)
RI4aV = ee.Geometry.LineString(
```

```
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-125.293972, -73.657177], [-125.495948, -73.713957], [-125.17542, -73.84897],
        [-124.644134, -73.817161], [-124.424595, -73.897795], [-124.108457, -73.987748],
        [-123.67816, -74.039788], [-123.291769, -73.851417], [-124.029423, -73.873401],
        [-124.34556, -73.844081], [-124.613399, -73.745969], [-124.775858, -73.729974],
        [-124.789031, -73.694232], [-125.293972, -73.657177]]
)
RI4bV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-125.255622, -73.425251], [-125.273185, -73.427132], [-125.258366,
        -73.430421], [-125.217751, -73.432458], [-125.168355, -73.430421], [-125.139815,
        -73.430108], [-125.106884, -73.430578], [-125.144206, -73.426191], [-125.233668,
        -73.426661], [-125.255622, -73.425251]]
)
RI4cV = ee.Geometry.LineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = [[-124.90381, -73.269408], [-124.916982, -73.27178], [-124.919177, -73.281422],
        [-124.899419, -73.297533], [-124.869781, -73.294849], [-124.890088, -73.279052],
        [-124.90381, -73.269408]]
)
RI4V = ee.Geometry.MultiLineString(
    geodesic = False,
    proj = 'EPSG:4326',
    coords = ee.List([RI4aV,RI4bV,RI4cV])
)

## (3.4) Validation Feature Collection
ValidationRings = ee.FeatureCollection([
    ee.Feature(OW1V,{
        'classes': 1,
        'subclasses': 1,
        'name': 'Open Water',
        'type': 'Water',
        'image': 'V1',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(0)).get('system:
            time_start'),
    }),
    ee.Feature(DW1V,{
        'classes': 1,
        'subclasses': 2,
        'name': 'Rough Water',
        'type': 'Water',
        'image': 'V1',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(0)).get('system:
            time_start'),
    }),
    ee.Feature(FI1V,{
        'classes': 2,
        'subclasses': 3,
        'name': 'Flat Sea Ice',
        'type': 'Ice',
        'image': 'V1',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(0)).get('system:
            time_start')
    }),
    ee.Feature(RI1V,{
        'classes': 2,
        'subclasses': 4,
        'name': 'Floating Land Ice',
        'type': 'Ice',
        'image': 'V1',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(0)).get('system:
            time_start')
    }),

    ee.Feature(OW2V,{
        'classes': 1,
        'subclasses': 1,
```

```
        'name': 'Open Water',
        'type': 'Water',
        'image': 'V2',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(1)).get('system:
            time_start'),
    }),
    ee.Feature(DW2V,{
        'classes': 1,
        'subclasses': 2,
        'name': 'Rough Water',
        'type': 'Water',
        'image': 'V2',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(1)).get('system:
            time_start'),
    }),
    ee.Feature(FI2V,{
        'classes': 2,
        'subclasses': 3,
        'name': 'Flat Sea Ice',
        'type': 'Ice',
        'image': 'V2',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(1)).get('system:
            time_start')
    }),
    ee.Feature(RI2V,{
        'classes': 2,
        'subclasses': 4,
        'name': 'Floating Land Ice',
        'type': 'Ice',
        'image': 'V2',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(1)).get('system:
            time_start')
    }),

    ee.Feature(OW3V,{
        'classes': 1,
        'subclasses': 1,
        'name': 'Open Water',
        'type': 'Water',
        'image': 'V3',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(2)).get('system:
            time_start'),
    }),
    ee.Feature(DW3V,{
        'classes': 1,
        'subclasses': 2,
        'name': 'Rough Water',
        'type': 'Water',
        'image': 'V3',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(2)).get('system:
            time_start'),
    }),
    ee.Feature(FI3V,{
        'classes': 2,
        'subclasses': 3,
        'name': 'Flat Sea Ice',
        'type': 'Ice',
        'image': 'V3',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(2)).get('system:
            time_start')
    }),
    ee.Feature(RI3V,{
        'classes': 2,
        'subclasses': 4,
        'name': 'Floating Land Ice',
        'type': 'Ice',
        'image': 'V3',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(2)).get('system:
            time_start')
    }),

```

```
    ee.Feature(OW4V,{
        'classes': 1,
        'subclasses': 1,
        'name': 'Open Water',
        'type': 'Water',
        'image': 'V4',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(3)).get('system:time_start'),
    }),
    ee.Feature(DW4V,{
        'classes': 1,
        'subclasses': 2,
        'name': 'Rough Water',
        'type': 'Water',
        'image': 'V4',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(3)).get('system:time_start'),
    }),
    ee.Feature(FI4V,{
        'classes': 2,
        'subclasses': 3,
        'name': 'Flat Sea Ice',
        'type': 'Ice',
        'image': 'V4',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(3)).get('system:time_start')
    }),
    ee.Feature(RI4V,{
        'classes': 2,
        'subclasses': 4,
        'name': 'Floating Land Ice',
        'type': 'Ice',
        'image': 'V4',
        'utc': ee.Image(ValidationData.toList(ValidationData.size()).get(3)).get('system:time_start')
    }),
])

ValidationPolys = ValidationRings.map(lambda feature: ee.Feature(ee.Geometry.MultiPolygon(feature.geometry().coordinates())).copyProperties(feature))

## (3.5) Visualization of Polygon Choices
### (3.5.1) Visualization Function
def mapRDPolys(S1,Opt,polygons,region):
    geometry = S1.first().geometry()

    Map = geemap.Map()
    Map.centerObject(geometry,6)

    RDList = S1.toList(S1.size())

    for i in range(RDList.size().getInfo()):
        image = ee.Image(RDList.get(i))

        matchIndex = ee.Number(image.get('match'))

        OptCol = Opt.filterMetadata('match','equals',matchIndex)
        OptList = OptCol.toList(OptCol.size())
        GeomList = OptList.map(lambda image: ee.Image(image).geometry())

        def combineGeometries(geom2,geom1):
            geom = ee.Geometry(geom1).union(ee.Geometry(geom2),maxError=1e3)
            return geom

        OptGeometry = ee.Geometry(GeomList.iterate(
            function = combineGeometries,
            first = ee.Geometry(GeomList.get(0))
        ))
        S1Geometry = image.geometry()
        OptImage = OptCol.median().clip(S1Geometry)
        S1Image = image.clip(OptGeometry)
```

```

        Map.addLayer(S1Image.select('HH'),S1_HH_params,'HH'+str(i+1))
        print('S1 HH image '+str(i+1)+' added')
        Map.addLayer(S1Image.select('HV'),S1_HV_params,'HV'+str(i+1))
        print('S1 HV image '+str(i+1)+' added')

        Map.addLayer(OptImage.select(RGBbands),RGB_params,'Opt'+str(i+1))
        print('Opt image '+str(i+1)+' added')

        polyCol = polygons.filter(ee.Filter.eq('utc',S1Image.get('system:time_start')))
        classes = polyCol.aggregate_array('name').getInfo()
        for k in range(len(classes)):
            polygon = polyCol.filter(ee.Filter.eq('name',classes[k]))
            if classes[k] == 'Open Water':
                color = 'Orange'
            elif classes[k] == 'Disturbed Water':
                color = 'Yellow'
            elif classes[k] == 'Flat Ice':
                color = 'Green'
            else:
                color = 'Cyan'
            Map.addLayer(polygon,{'color': color},classes[k]+' ('+color+')_'+str(i+1))
            print(classes[k]+' polygon added')

    html_file = os.path.join(html_dir+'\Polygons\Training_Validation\_'+region+'.html')
    Map.to_html(outfile=html_file, title= 'Polygon Choices', width='100%', height='880px')

    print('Map finished')

    return Map, Map

## Leave commented!!
## Used to visualize data set

# PolygonMap, Map = mapRDPolys(S1Training,OptTraining,TrainingRings,'Training')
# PolygonMap, Map = mapRDPolys(S1Validation,OptValidation,ValidationRings,'Validation')
# Map

#%% (4) Sampling
## (4.1) Functions
def extractPixelValues(data,multiPolygons,numPixels):
    dataList = data.toList(data.size())

    def samplePerImage(image,multiPolygons,numPixels):
        utc = image.get('system:time_start')
        multiPolygon = multiPolygons.filterMetadata('utc','equals',utc)
        names = multiPolygon.aggregate_array('name')

        def samplePerPolygon(image,multiPolygon,name,numPixels):
            region = multiPolygon.filterMetadata('name','equals',name).geometry()
            polygons = region.geometries()

            subSampled =  polygons
                .map(lambda geometry: image                    .sample(
                    region = ee.Geometry(geometry),
                    scale = image.select(0).projection().nominalScale(),
                    projection = image.select(0).projection().crs(),
                    numPixels = ee.Number(numPixels).divide(polygons.size()).ceil(),
                    seed = 1,
                    geometries = True)\
                .map(lambda feat: feat.copyProperties(multiPolygon.filterMetadata('name','equals',name).first()))
                )

            sampledFC = ee.FeatureCollection(subSampled).flatten()

            return sampledFC

        sampledList = names.map(lambda name: samplePerPolygon(image,multiPolygon,name,numPixels))
        sampled = ee.FeatureCollection(sampledList).flatten()
```

```
        return sampled

    sampleList = dataList.map(lambda image: samplePerImage(ee.Image(image),multiPolygons,
        numPixels))
    sampleCol = ee.FeatureCollection(sampleList).flatten()

    return sampleCol

def extractPixelValuesSubPolys(data,Polygons,numPixels):
    dataList = data.toList(data.size())

    def samplePerImage(image,Polygons,numPixels):
        utc = image.get('system:time_start')
        Polygon = Polygons.filterMetadata('utc','equals',utc)
        names = Polygon.aggregate_array('name')

        def samplePerPolygon(image,Polygon,name,numPixels):
            region = Polygon.filterMetadata('name','equals',name).geometry()

            sampledFC =  image.sample(
                    region = region,
                    scale = image.select(0).projection().nominalScale(),
                    projection = image.select(0).projection().crs(),
                    numPixels = numPixels,
                    seed = 1,
                    geometries = True)\
                .map(lambda feat: feat.copyProperties(Polygon.filterMetadata('name','equals',
                    name).first()))

            return sampledFC

        sampledList = names.map(lambda name: samplePerPolygon(image,Polygons,name,numPixels))
        sampled = ee.FeatureCollection(sampledList).flatten()

        return sampled

    sampleList = dataList.map(lambda image: samplePerImage(ee.Image(image),Polygons,numPixels
        ))
    sampleCol = ee.FeatureCollection(sampleList).flatten()

    return sampleCol

def clipToPolygon(image,polys,windowsize):
    polygons = polys.filterMetadata('utc','equals',image.get('system:time_start'))
    bufferwidth = ee.Number(windowsize).subtract(1).multiply(2)
    error = bufferwidth.divide(50)
    buffered = polygons.geometry().buffer(bufferwidth,maxError=error)
    clipped = image.clip(buffered)

    return clipped

def computeGLCM(image,band,windowsize,kernel):
    glcm = image.addBands(image.select(band).toInt32().glcmTexture(size=windowsize,kernel=
        kernel))

    return glcm

def clipToThresholds(image,band,upperlimit,lowerlimit):
    upperMask = image.gt(upperlimit)
    lowerMask = image.lt(lowerlimit)

    clippedUpper = upperMask.multiply(upperlimit).add(image.select(band).multiply(upperMask.
        Not()))
    clippedLower = lowerMask.multiply(lowerlimit).add(clippedUpper.multiply(lowerMask.Not()))

    return image.addBands(clippedLower.select(band),overwrite=True)

## (4.2) Computation
## NOTE: Computation is done during export to reduce computational effort
```

```
#%% (5) Export of Training & Validation Data
## (5.1) GLCM5
numPixels = 1000
size = 5

## Leave commented!!
## Export was necessary to create feature collections used in section below

# geemap.ee_export_vector_to_drive(
#     ee_object = extractPixelValues(
#         TrainingData\
#             .select([’HH’,’HV’])\
#             .map(lambda image: clipToPolygon(image,TrainingPolys,size))\
#             .map(lambda image: image.addBands(image.select(’HH’).toInt32().glcmTexture(size=size,kernel=None)))\
#             .map(lambda image: image.addBands(image.select(’HV’).toInt32().glcmTexture(size=size,kernel=None))),
#         TrainingPolys,
#         numPixels),
#     description = ’TrainingValues_w’+str(size)+’_’+str(int(numPixels))+’pixels’,
#     folder = ’PixelValues_w’+str(size)+’_’+str(int(numPixels))+’pixels’,
#     file_format = ’shp’)

# geemap.ee_export_vector_to_drive(
#     ee_object = extractPixelValues(
#         ValidationData\
#             .select([’HH’,’HV’])\
#             .map(lambda image: clipToPolygon(image,ValidationPolys,size))\
#             .map(lambda image: image.addBands(image.select(’HH’).toInt32().glcmTexture(size=size,kernel=None)))\
#             .map(lambda image: image.addBands(image.select(’HV’).toInt32().glcmTexture(size=size,kernel=None))),
#         ValidationPolys,
#         numPixels),
#     description = ’ValidationValues_w’+str(size)+’_’+str(int(numPixels))+’pixels’,
#     folder = ’PixelValues_w’+str(size)+’_’+str(int(numPixels))+’pixels’,
#     file_format = ’shp’)

## (5.2) GLCM11
numPixels = 1000
size = 11

## Leave commented!!
## Export was necessary to create feature collections used in section below

# TDList = TrainingData.toList(TrainingData.size())
# for i in range(TrainingData.size().getInfo()):
#     imCol = ee.ImageCollection(ee.Image(TDList.get(i)))
#     geemap.ee_export_vector_to_drive(
#         ee_object = extractPixelValues(
#             imCol\
#                 .select([’HH’,’HV’])\
#                 .map(lambda image: clipToPolygon(image,TrainingPolys,size))\
#                 .map(lambda image: image.addBands(image.select(’HH’).toInt32().glcmTexture(size=size,kernel=None)))\
#                 .map(lambda image: image.addBands(image.select(’HV’).toInt32().glcmTexture(size=size,kernel=None))),
#             TrainingPolys,
#             numPixels),
#         description = ’TrainingValues_w’+str(size)+’_’+str(int(numPixels))+’pixels_Img’+str(i+1),
#         folder = ’PixelValues_w’+str(size)+’_’+str(int(numPixels))+’pixels’,
#         file_format = ’shp’)

# geemap.ee_export_vector_to_drive(
#     ee_object = extractPixelValues(
#         ValidationData\
#             .select([’HH’,’HV’])\
#             .map(lambda image: clipToPolygon(image,ValidationPolys,size))\
#             .map(lambda image: image.addBands(image.select(’HH’).toInt32().glcmTexture(size=size,kernel=None)))\
```

```
#              .map(lambda image: image.addBands(image.select('HV').toInt32().glcmTexture(size
    =size,kernel=None))),
#          ValidationPolys,
#          numPixels),
#      description = 'ValidationValues_w'+str(size)+'_'+str(int(numPixels))+'pixels',
#      folder = 'PixelValues_w'+str(size)+'_'+str(int(numPixels))+'pixels',
#      file_format = 'shp')

## (5.3) GLCM21
### (5.3.1) Computation & Export of Data Subsets
## Leave commented!!
## Export was necessary to create feature collections used in section below

numPixels = 1000
size = 21

## Training Data
# pixelThreshold = 3e6

# TDList = TrainingData.toList(TrainingData.size())
# for i in range(TDList.size().getInfo()):
#     if not i == 5:
#         continue
#     numPixels = 1000
#     image = ee.Image(TDList.get(i)).select(['HH','HV'])
#     clipped = clipToPolygon(image,TrainingPolys,size)

#     MultiPolygons = TrainingPolys.filterMetadata('utc','equals',image.get('system:
    time_start'))

#     pixelCount = clipped.select(0).reduceRegion(
#         reducer = ee.Reducer.count(),
#         geometry = clipped.geometry(),
#         scale = image.select(0).projection().nominalScale(),
#         crs = image.select(0).projection().crs()
#         ).get(image.bandNames().get(0)).getInfo()
#     print('Image ',str(i+1), ', pixel number for all polygons: ',pixelCount)

#     if pixelCount > pixelThreshold:
#         PolyNames = ['Open Water','Rough Water','Flat Sea Ice','Floating Land Ice']
#         abbrev = ['OW','RW','FSI','FLI']
#         for j in range(len(PolyNames)):
#             if not abbrev[j] == 'RW':
#                 continue
#             MultiPolygon = MultiPolygons.filterMetadata('name','equals',PolyNames[j])
#             clipped = clipToPolygon(image,MultiPolygon,size)

#             pixelCount = clipped.select(0).reduceRegion(
#                 reducer = ee.Reducer.count(),
#                 geometry = clipped.geometry(),
#                 scale = image.select(0).projection().nominalScale(),
#                 crs = image.select(0).projection().crs()
#                 ).get(image.bandNames().get(0)).getInfo()
#             print('Pixel number for '+PolyNames[j]+' polygons: ',pixelCount)

#             if pixelCount > pixelThreshold:
#                 print('Subpolygons of one class')
#                 subPolys = MultiPolygon.geometry().geometries()\
#                     .map(lambda geom: ee.Feature(ee.Geometry(geom)).copyProperties(
    MultiPolygon.first()))
#                 numSubPolys = subPolys.size().getInfo()
#                 numPixelsOriginal = numPixels
#                 numPixels = np.ceil(numPixelsOriginal/numSubPolys)
#                 for k in range(numSubPolys):
#                     if k == 0:
#                         continue
#                     Polygon = ee.FeatureCollection(ee.Feature(subPolys.get(k)))
#                     clipped = clipToPolygon(image,Polygon,size)

#                     pixelCount = clipped.select(0).reduceRegion(
#                         reducer = ee.Reducer.count(),
```

```
#                        geometry = clipped.geometry(),
#                        scale = image.select(0).projection().nominalScale(),
#                        crs = image.select(0).projection().crs()
#                        ).get(image.bandNames().get(0)).getInfo()
#                    print('Pixel number for '+PolyNames[j]+' subpolygon '+str(k+1)+': ', pixelCount)

#                    if pixelCount > pixelThreshold:
#                        shrink = -1*np.sqrt(pixelCount)/4*40
#                        smallGeometry = Polygon.geometry().buffer(shrink,maxError=1e3)
#                        smallPolygon = ee.FeatureCollection([ee.Feature(smallGeometry).copyProperties(MultiPolygon.first())])
#                        clipped = clipToPolygon(image,smallPolygon,size)

#                        pixelCount = clipped.select(0).reduceRegion(
#                            reducer = ee.Reducer.count(),
#                            geometry = clipped.geometry(),
#                            scale = image.select(0).projection().nominalScale(),
#                            crs = image.select(0).projection().crs()
#                            ).get(image.bandNames().get(0)).getInfo()
#                        print('Pixel number for '+PolyNames[j]+' subpolygon '+str(k+1)+' (shrinked): ',pixelCount)

#                        if smallGeometry.type().getInfo() == 'Polygon':
#                            geemap.ee_export_vector_to_drive(
#                                ee_object = extractPixelValuesSubPolys(
#                                    ee.ImageCollection(clipped)\
#                                        .select(['HH','HV'])\
#                                        .map(lambda image: image.addBands(image.select('HH').toInt32().glcmTexture(size=size,kernel=None)))\
#                                        .map(lambda image: image.addBands(image.select('HV').toInt32().glcmTexture(size=size,kernel=None))),
#                                    smallPolygon,
#                                    numPixels),
#                                description = 'TrainingValues_w'+str(size)+'_'+str(int(numPixelsOriginal))+'pixels_Img'+str(i+1)+'_'+abbrev[j]+'_'+str(k+1)+'_shrinked',
#                                folder = 'PixelValues_w'+str(size)+'_'+str(int(numPixelsOriginal))+'pixels',
#                                file_format = 'shp')
#                        else:
#                            geemap.ee_export_vector_to_drive(
#                                ee_object = extractPixelValues(
#                                    ee.ImageCollection(clipped)\
#                                        .select(['HH','HV'])\
#                                        .map(lambda image: image.addBands(image.select('HH').toInt32().glcmTexture(size=size,kernel=None)))\
#                                        .map(lambda image: image.addBands(image.select('HV').toInt32().glcmTexture(size=size,kernel=None))),
#                                    smallPolygon,
#                                    numPixels),
#                                description = 'TrainingValues_w'+str(size)+'_'+str(int(numPixelsOriginal))+'pixels_Img'+str(i+1)+'_'+abbrev[j]+'_'+str(k+1)+'_shrinked',
#                                folder = 'PixelValues_w'+str(size)+'_'+str(int(numPixelsOriginal))+'pixels',
#                                file_format = 'shp')

#                    else:
#                        geemap.ee_export_vector_to_drive(
#                            ee_object = extractPixelValues(
#                                ee.ImageCollection(clipped)\
#                                    .select(['HH','HV'])\
#                                    .map(lambda image: image.addBands(image.select('HH').toInt32().glcmTexture(size=size,kernel=None)))\
#                                    .map(lambda image: image.addBands(image.select('HV').toInt32().glcmTexture(size=size,kernel=None))),
#                                Polygon,
#                                numPixels),
#                            description = 'TrainingValues_w'+str(size)+'_'+str(int(numPixelsOriginal))+'pixels_Img'+str(i+1)+'_'+abbrev[j]+'_'+str(k+1),
#                            folder = 'PixelValues_w'+str(size)+'_'+str(int(numPixelsOriginal))+'pixels',
```

```
#                             file_format = 'shp')
#             else:
#                 geemap.ee_export_vector_to_drive(
#                     ee_object = extractPixelValues(
#                         ee.ImageCollection(clipped)\
#                             .select(['HH','HV'])\
#                             .map(lambda image: image.addBands(image.select('HH').toInt32().glcmTexture(size=size,kernel=None)))\
#                             .map(lambda image: image.addBands(image.select('HV').toInt32().glcmTexture(size=size,kernel=None))),
#                         MultiPolygon,
#                         numPixels),
#                     description = 'TrainingValues_w'+str(size)+'_'+str(int(numPixels))+'pixels_Img'+str(i+1)+'_'+abbrev[j],
#                     folder = 'PixelValues_w'+str(size)+'_'+str(int(numPixels))+'pixels',
#                     file_format = 'shp')
#     else:
#         geemap.ee_export_vector_to_drive(
#             ee_object = extractPixelValues(
#                 ee.ImageCollection(clipped)\
#                     .select(['HH','HV'])\
#                     .map(lambda image: image.addBands(image.select('HH').toInt32().glcmTexture(size=size,kernel=None)))\
#                     .map(lambda image: image.addBands(image.select('HV').toInt32().glcmTexture(size=size,kernel=None))),
#                 TrainingPolys,
#                 numPixels),
#             description = 'TrainingValues_w'+str(size)+'_'+str(int(numPixels))+'pixels_Img'+str(i+1)+'_All',
#             folder = 'PixelValues_w'+str(size)+'_'+str(int(numPixels))+'pixels',
#             file_format = 'shp')

## Validation Data
# pixelThreshold = 5e6

# VDList = ValidationData.toList(ValidationData.size())
# for i in range(VDList.size().getInfo()):
#     if not i == 2:
#         continue
#     numPixels = 1000
#     image = ee.Image(VDList.get(i)).select(['HH','HV'])
#     clipped = clipToPolygon(image,ValidationPolys,size)

#     MultiPolygons = ValidationPolys.filterMetadata('utc','equals',image.get('system:time_start'))

#     pixelCount = clipped.select(0).reduceRegion(
#         reducer = ee.Reducer.count(),
#         geometry = clipped.geometry(),
#         scale = image.select(0).projection().nominalScale(),
#         crs = image.select(0).projection().crs()
#         ).get(image.bandNames().get(0)).getInfo()
#     print('Image ',str(i+1), ', pixel number for all polygons: ',pixelCount)

#     if pixelCount > pixelThreshold:
#         PolyNames = ['Open Water','Rough Water','Flat Sea Ice','Floating Land Ice']
#         abbrev = ['OW','RW','FSI','FLI']
#         for j in range(len(PolyNames)):
#             MultiPolygon = MultiPolygons.filterMetadata('name','equals',PolyNames[j])
#             clipped = clipToPolygon(image,MultiPolygon,size)

#             pixelCount = clipped.select(0).reduceRegion(
#                 reducer = ee.Reducer.count(),
#                 geometry = clipped.geometry(),
#                 scale = image.select(0).projection().nominalScale(),
#                 crs = image.select(0).projection().crs()
#                 ).get(image.bandNames().get(0)).getInfo()
#             print('Pixel number for '+PolyNames[j]+' polygons: ',pixelCount)

#             if pixelCount > pixelThreshold:
#                 print('Subpolygons of one class')
```

```
#                 subPolys = MultiPolygon.geometry().geometries()\
#                     .map(lambda geom: ee.Feature(ee.Geometry(geom)).copyProperties(MultiPolygon.first()))
#                 numSubPolys = subPolys.size().getInfo()
#                 for k in range(numSubPolys):
#                     numPixels = np.ceil(numPixels/numSubPolys)
#                     Polygon = ee.FeatureCollection(ee.Feature(subPolys.get(k)))
#                     clipped = clipToPolygon(image,Polygon,size)

#                     pixelCount = clipped.select(0).reduceRegion(
#                         reducer = ee.Reducer.count(),
#                         geometry = clipped.geometry(),
#                         scale = image.select(0).projection().nominalScale(),
#                         crs = image.select(0).projection().crs()
#                         ).get(image.bandNames().get(0)).getInfo()
#                     print(’Pixel number for ’+PolyNames[j]+’ subpolygon ’+str(k+1)+’: ’,pixelCount)

#                     if pixelCount > pixelThreshold:
#                         shrink = -1*np.sqrt(pixelCount)/8*40
#                         smallGeometry = Polygon.geometry().buffer(shrink,maxError=1e3)
#                         smallPolygon = ee.FeatureCollection([ee.Feature(smallGeometry).copyProperties(MultiPolygon.first())])
#                         clipped = clipToPolygon(image,smallPolygon,size)

#                         pixelCount = clipped.select(0).reduceRegion(
#                             reducer = ee.Reducer.count(),
#                             geometry = clipped.geometry(),
#                             scale = image.select(0).projection().nominalScale(),
#                             crs = image.select(0).projection().crs()
#                             ).get(image.bandNames().get(0)).getInfo()
#                         print(’Pixel number for ’+PolyNames[j]+’ subpolygon ’+str(k+1)+’ (shrinked): ’,pixelCount)

#                         if smallGeometry.type().getInfo() == ’Polygon’:
#                             geemap.ee_export_vector_to_drive(
#                                 ee_object = extractPixelValuesSubPolys(
#                                     ee.ImageCollection(clipped)\
#                                         .select([’HH’,’HV’])\
#                                         .map(lambda image: image.addBands(image.select(’HH’).toInt32().glcmTexture(size=size,kernel=None)))\
#                                         .map(lambda image: image.addBands(image.select(’HV’).toInt32().glcmTexture(size=size,kernel=None))),
#                                     smallPolygon,
#                                     numPixels),
#                                 description = ’ValidationValues_w’+str(size)+’_’+str(int(numPixels))+’pixels_Img’+str(i+1)+’_’+abbrev[j]+’_’+str(k+1)+’_shrinked’,
#                                 folder = ’PixelValues_w’+str(size)+’_’+str(int(numPixels))+’pixels’,
#                                 file_format = ’shp’)
#                         else:
#                             geemap.ee_export_vector_to_drive(
#                                 ee_object = extractPixelValues(
#                                     ee.ImageCollection(clipped)\
#                                         .select([’HH’,’HV’])\
#                                         .map(lambda image: image.addBands(image.select(’HH’).toInt32().glcmTexture(size=size,kernel=None)))\
#                                         .map(lambda image: image.addBands(image.select(’HV’).toInt32().glcmTexture(size=size,kernel=None))),
#                                     smallPolygon,
#                                     numPixels),
#                                 description = ’ValidationValues_w’+str(size)+’_’+str(int(numPixels))+’pixels_Img’+str(i+1)+’_’+abbrev[j]+’_’+str(k+1)+’_shrinked’,
#                                 folder = ’PixelValues_w’+str(size)+’_’+str(int(numPixels))+’pixels’,
#                                 file_format = ’shp’)

#                     else:
#                         geemap.ee_export_vector_to_drive(
#                             ee_object = extractPixelValues(
#                                 ee.ImageCollection(clipped)\
```

```
#                                     .select(['HH','HV'])\
#                                     .map(lambda image: image.addBands(image.select('HH').toInt32().glcmTexture(size=size,kernel=None)))\
#                                     .map(lambda image: image.addBands(image.select('HV').toInt32().glcmTexture(size=size,kernel=None))),
#                                 Polygon,
#                                 numPixels),
#                             description = 'ValidationValues_w'+str(size)+'_'+str(int(numPixels))+'pixels_Img'+str(i+1)+'_'+abbrev[j]+'_'+str(k+1),
#                             folder = 'PixelValues_w'+str(size)+'_'+str(int(numPixels))+'pixels',
#                             file_format = 'shp')
#             else:
#                 geemap.ee_export_vector_to_drive(
#                     ee_object = extractPixelValues(
#                         ee.ImageCollection(clipped)\
#                             .select(['HH','HV'])\
#                             .map(lambda image: image.addBands(image.select('HH').toInt32().glcmTexture(size=size,kernel=None)))\
#                             .map(lambda image: image.addBands(image.select('HV').toInt32().glcmTexture(size=size,kernel=None))),
#                         MultiPolygon,
#                         numPixels),
#                     description = 'ValidationValues_w'+str(size)+'_'+str(int(numPixels))+'pixels_Img'+str(i+1)+'_'+abbrev[j],
#                     folder = 'PixelValues_w'+str(size)+'_'+str(int(numPixels))+'pixels',
#                     file_format = 'shp')
#     else:
#         geemap.ee_export_vector_to_drive(
#             ee_object = extractPixelValues(
#                 ee.ImageCollection(clipped)\
#                     .select(['HH','HV'])\
#                     .map(lambda image: image.addBands(image.select('HH').toInt32().glcmTexture(size=size,kernel=None)))\
#                     .map(lambda image: image.addBands(image.select('HV').toInt32().glcmTexture(size=size,kernel=None))),
#                 ValidationPolys,
#                 numPixels),
#             description = 'ValidationValues_w'+str(size)+'_'+str(int(numPixels))+'pixels_Img'+str(i+1)+'_All',
#             folder = 'PixelValues_w'+str(size)+'_'+str(int(numPixels))+'pixels',
#             file_format = 'shp')

### (5.3.2) Import Sampled Data Subsets & Export as Full Data Set
## Convert feature collections per image to full feature collection for training/validation data sets:
## NOTE: edit earth engine name to that of personal account if you want to redo this process

## GLCM11:
# numPixels = 1000
# size = 11

# TV11_list = []
# for i in range(S1Training.size().getInfo()):
#     fc = ee.FeatureCollection('users/skjeltmaps/TrainingValues_w11_1000pixels_Img'+str(i+1))

# TV11 = ee.FeatureCollection(TV11_list).flatten()

# geemap.ee_export_vector_to_drive(
#     ee_object = TV11,
#     description = 'TrainingValues_w11_1000pixels',
#     folder = 'PixelValues_w11_1000pixels',
#     file_format = 'shp')

## GLCM21:
# numPixels = 1000
# size = 21

# trainNames = [
#     'TrainingValues_w21_1000pixels_Img1_All',
```

```
#     'TrainingValues_w21_1000pixels_Img2_All',
#     'TrainingValues_w21_1000pixels_Img3_All',
#     'TrainingValues_w21_1000pixels_Img4_All',
#     'TrainingValues_w21_1000pixels_Img5_All',
#     'TrainingValues_w21_1000pixels_Img6_FSI',
#     'TrainingValues_w21_1000pixels_Img6_FLI',
#     'TrainingValues_w21_1000pixels_Img6_OW',
#     'TrainingValues_w21_1000pixels_Img6_RW_1_shrinked',
#     'TrainingValues_w21_1000pixels_Img6_RW_2',
#     'TrainingValues_w21_1000pixels_Img6_RW_3',
#     'TrainingValues_w21_1000pixels_Img6_RW_4',
#     'TrainingValues_w21_1000pixels_Img6_RW_5',
#     'TrainingValues_w21_1000pixels_Img6_RW_6',
#     'TrainingValues_w21_1000pixels_Img6_RW_7',
#     'TrainingValues_w21_1000pixels_Img7_All',
#     'TrainingValues_w21_1000pixels_Img8_All',
#     'TrainingValues_w21_1000pixels_Img9_All'
# ]

# TV21_list = []
# for i in range(len(trainNames)):
#     fc = ee.FeatureCollection('users/skjeltmaps/'+trainNames[i])
#     TV21_list.append(fc)
# TV21 = ee.FeatureCollection(TV21_list).flatten()

# geemap.ee_export_vector_to_drive(
#     ee_object = TV21,
#     description = 'TrainingValues_w21_1000pixels',
#     folder = 'PixelValues_w21_1000pixels',
#     file_format = 'shp')

# valNames = [
#     'ValidationValues_w21_1000pixels_Img1_All',
#     'ValidationValues_w21_1000pixels_Img2_All',
#     'ValidationValues_w21_1000pixels_Img3_FSI',
#     'ValidationValues_w21_1000pixels_Img3_FLI',
#     'ValidationValues_w21_1000pixels_Img3_OW',
#     'ValidationValues_w21_1000pixels_Img3_RW',
#     'ValidationValues_w21_1000pixels_Img4_All',
# ]

# VV21_list = []
# for i in range(len(valNames)):
#     featcol = ee.FeatureCollection('users/skjeltmaps/'+valNames[i])
#     VV21_list.append(featcol)
# VV21 = ee.FeatureCollection(VV21_list).flatten()

# geemap.ee_export_vector_to_drive(
#     ee_object = VV21,
#     description = 'ValidationValues_w21_1000pixels',
#     folder = 'PixelValues_w21_1000pixels',
#     file_format = 'shp')
```

B.6. Feature Selection

6_FeatureSelection

```
#%% (1) Code set-up
## (1.1) Import packages
import ee
import geemap
import numpy as np
import os
import matplotlib.pyplot as plt
import scipy.stats as stats

from sklearn.feature_selection import RFECV
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold
```

```python
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV

## (1.2) Initialize
ee.Initialize()

## (1.3) Parameterization & Data Import
SMALL_SIZE = 24
MEDIUM_SIZE = 28
BIGGER_SIZE = 32

plt.rc('font', size=SMALL_SIZE)          # controls default text sizes
plt.rc('axes', titlesize=SMALL_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=SMALL_SIZE)    # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE)  # fontsize of the figure title

properties = ['','_asm','_contrast','_corr','_dent','_diss','_dvar','_ent','_idm','_imcorr1','_imcorr2','_inertia','_prom',
              '_savg','_sent','_shade','_svar','_var']
names = ['Intensity','Angular Second Moment','Contrast','Correlation','Difference Entropy','Dissimilarity',
         'Difference Variance','Entropy','Inverse Distance Moment','Information Measure 1 of Correlation',
         'Information Measure 2 of Correlation','Inertia','Cluster Prominence','Sum Average','Sum Entropy','Cluster Shade',
         'Sum Variance','Variance']

property_names = []
feature_names = []
for i in range(len(properties)):
    name = names[i]
    prop = properties[i]

    feature_names.append(name+' (HH)')
    feature_names.append(name+' (HV)')

    property_names.append('HH'+prop)
    property_names.append('HV'+prop)

TV11 = ee.FeatureCollection('users/skjeltmaps/TrainingValues_w11_1000pixels')  .map(lambda feat: feat.set('HH_contrast',feat.get('HH_contras')).set('HV_contrast',feat.get('HV_contras')))
TV5 = ee.FeatureCollection('users/skjeltmaps/TrainingValues_w5_1000pixels')  .map(lambda feat: feat.set('HH_contrast',feat.get('HH_contras')).set('HV_contrast',feat.get('HV_contras')))
TV21 = ee.FeatureCollection('users/skjeltmaps/TrainingValues_w21_1000pixels')  .map(lambda feat: feat.set('HH_contrast',feat.get('HH_contras')).set('HV_contrast',feat.get('HV_contras')))
TV0 = TV11.select(['HH','HV','classes','subclasses','name','utc','type','image'])

VV11 = ee.FeatureCollection('users/skjeltmaps/ValidationValues_w11_1000pixels')  .map(lambda feat: feat.set('HH_contrast',feat.get('HH_contras')).set('HV_contrast',feat.get('HV_contras')))
VV5 = ee.FeatureCollection('users/skjeltmaps/ValidationValues_w5_1000pixels')  .map(lambda feat: feat.set('HH_contrast',feat.get('HH_contras')).set('HV_contrast',feat.get('HV_contras')))
VV21 = ee.FeatureCollection('users/skjeltmaps/ValidationValues_w21_1000pixels')  .map(lambda feat: feat.set('HH_contrast',feat.get('HH_contras')).set('HV_contrast',feat.get('HV_contras')))
VV0 = VV11.select(['HH','HV','classes','subclasses','name','utc','type','image'])

#%% (2) Retrieve Sample Data
## (2.1) Function
def SampleDataClientSide(eeTraining,eeValidation,property_names,n_features):
    n_samples_training = eeTraining.size().getInfo()
    train_data = np.empty([n_samples_training,n_features])

    for i in range(n_features):
```

```
        train_data[:,i] = eeTraining.aggregate_array(property_names[i]).getInfo()

    train_classes = eeTraining.aggregate_array('classes').getInfo()
    train_subclasses = eeTraining.aggregate_array('subclasses').getInfo()

    return [train_data, train_classes, train_subclasses]

## (2.2) Download Data to Client Side
sample_data_glcm5 = SampleDataClientSide(TV5,VV5,property_names,len(property_names))
sample_data_glcm11 = SampleDataClientSide(TV11,VV11,property_names,len(property_names))
sample_data_glcm21 = SampleDataClientSide(TV21,VV21,property_names,len(property_names))

#%% (3) Correlation
## (3.1) Function
def heavilyCorrelated(data,property_names,threshold):
    corr = np.corrcoef(np.array(data),rowvar=False)
    pos_corr = np.logical_and(corr > threshold,corr < 0.999)
    neg_corr = np.logical_and(corr < -threshold,corr > -0.999)
    boolean = np.logical_or(pos_corr,neg_corr)
    correlated_names = []
    correlated = []
    for i in range(len(data[0,:])):
        corr_names = [corrs for corrs,check in zip(property_names, boolean[i]) if check]
        correlated_names.append(corr_names)

        corr_values = [vals for vals,check in zip(corr[i], boolean[i]) if check]
        correlated.append(corr_values)

        print(property_names[i]+': '+str(corr_names)+', values: '+str(corr_values))
    return [correlated_names, correlated]

## (3.2) Compute Correlation Between Features
correlations = heavilyCorrelated(sample_data_glcm11[0],property_names,0.9)

#%% (4) Hyperparameter Test
# Number of trees in random forest
n_estimators = np.arange(10,100)
# Maximum number of levels in tree
max_depth = np.arange(1,30)
# Minimum number of samples required to split a node
min_samples_split = np.arange(1,10)
# Minimum number of leafs required to be at a leaf node
min_samples_leaf = np.arange(1,10)
# Create the random grid
parameters = [n_estimators,max_depth,min_samples_split,min_samples_leaf]
parameter_names = ['n_estimators','max_depth','min_samples_split','min_samples_leaf']
random_grid = {'n_estimators': n_estimators,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf}

rf_base = RandomForestClassifier()
rf_random = RandomizedSearchCV(estimator = rf_base, param_distributions = random_grid,
                               n_iter = 200, cv = 3, verbose = 2, random_state = 1,
                               n_jobs = -1)
rf_grid = GridSearchCV(estimator = rf_base, param_grid = random_grid,
                               cv = 3, verbose = 2, n_jobs = -1)

# Fit the random search model
# rf_random.fit(sample_data_glcm11[0][:,:2], sample_data_glcm11[1])
# rf_grid.fit(training_data0, training_labels0)
# View the best parameters from the random search
# print(rf_random.best_params_)
# {'n_estimators': 75, 'min_samples_split': 3, 'min_samples_leaf': 1, 'max_depth': 3}

rf = RandomForestClassifier(
    n_estimators = 10,
    max_depth = None,
    min_samples_split = 2,
    min_samples_leaf = 1,
    random_state = 1
```

```
)

#%% (5) Recursive Feature Elimination with Cross Validation
## (5.1) Functions
def RFE_CV(rf,w,feature_names,classification_label,visualize=True):

    sampledata = eval('sample_data_glcm'+str(w))
    training_data = sampledata[0]
    if classification_label == 'classes':
        labels = sampledata[1]
        classification_type = 'Water/Ice'
    if classification_label == 'subclasses':
        labels = sampledata[2]
        classification_type = 'OW/RW/SI/FLI'

    rfecv = RFECV(estimator=rf, step=1, cv=StratifiedKFold(3), scoring='accuracy')
    rfecv.fit(training_data,labels)

    features = [f for f,s in zip(feature_names, rfecv.support_) if s]
    accuracies = np.mean(rfecv.grid_scores_,axis=1)

    if visualize:
        titlename = '{} Classification, '.format(classification_type)+'including GLCM textures (w={})'.format(str(w))

        plt.figure(figsize=(16, 9))
        plt.title(titlename, fontweight='bold', pad=20)
        plt.xlabel('Number of selected features', labelpad=20)
        plt.ylabel('Accuracy [-]', labelpad=20)
        plt.plot(range(1,len(feature_names)+1), accuracies, color='#303F9F', linewidth=3)
        plt.xlim(1,len(feature_names))
        plt.grid()
        plt.show()

    return features, accuracies

def FeatureImportanceScores(rf,w,feature_names,best_features,classification_label,visualize=True):
    sampledata = eval('sample_data_glcm'+str(w))
    training_data = sampledata[0]
    if classification_label == 'classes':
        labels = sampledata[1]
        classification_type = 'Water/Ice'
    if classification_label == 'subclasses':
        labels = sampledata[2]
        classification_type = 'OW/RW/SI/FLI'

    best_indeces = []
    for i in range(len(best_features)):
        best_indeces.append(np.argwhere(np.array(feature_names) == best_features[i])[0][0])
    best_feature_data = training_data[:,best_indeces]

    fitted = rf.fit(best_feature_data,labels)
    importances = fitted.feature_importances_
    sorted_indeces = importances.argsort()
    features_sorted = [best_features[index] for index in sorted_indeces]
    importances_sorted = np.sort(importances)

    if visualize:
        titlename = '{} Classification, '.format(classification_type)+'including GLCM textures (w={})'.format(str(w))

        fig, ax = plt.subplots(figsize=(16, 9))
        ax.xaxis.grid(True)
        ax.barh(features_sorted,importances_sorted)

        plt.title(titlename, fontweight='bold', pad=20)
        plt.ylabel('Feature name', labelpad=20)
        plt.xlabel('Feature importance score [-]')
        plt.show()

```

```
    return importances

def StatisticalSignificance(rf,w,feature_names,classification_label,visualize=True):
    sampledata = eval('sample_data_glcm'+str(w))
    training_data = sampledata[0]
    if classification_label == 'classes':
        labels = sampledata[1]
        classification_type = 'Water/Ice'
    if classification_label == 'subclasses':
        labels = sampledata[2]
        classification_type = 'OW/RW/SI/FLI'

    predicted_labels = []
    best_features_list = []

    feature_iterations = np.arange(2,len(feature_names)+1)
    for _,n_features in enumerate(feature_iterations):

        rfe = RFE(estimator=rf, n_features_to_select=n_features, step=1)
        fitted_all = rfe.fit(training_data,labels)

        best_features = [f for f,s in zip(feature_names, fitted_all.support_) if s]
        best_indeces = []
        for i in range(len(best_features)):
            best_indeces.append(np.argwhere(np.array(feature_names) == best_features[i])
                [0][0])
        best_feature_data = training_data[:,best_indeces]

        fitted_best = rfe.fit(best_feature_data,labels)
        predicted_labels.append(fitted_best.predict(best_feature_data))

    pvalues = [0]
    for i in range(len(predicted_labels)):
        _, pval = stats.ttest_ind(labels,predicted_labels[i])
        pvalues.append(pval)

    if visualize:
        titlename = 'T-test p-values of predicted vs actual class labels, including GLCM
            textures (w={})'.format(str(w))

        plt.figure(figsize=(16, 9))
        plt.title(titlename, fontweight='bold', pad=20)
        plt.xlabel('Number of selected features', labelpad=20)
        plt.ylabel('p-value [-]', labelpad=20)
        plt.plot(range(1, len(feature_names) + 1), pvalues, color='#303F9F', linewidth=3)
        plt.hlines(0.05,1,len(feature_names),colors='#303F9F',linestyles='dashed')
        plt.legend(['p-values','Significance threshold'])

        plt.xlim(1,len(feature_names))
        plt.ylim(bottom=0)
        plt.grid(axis='x')
        plt.show()

    return pvalues

def FeatureSelectionFigure(w,classification_label,feature_names,rfecv_features,rfecv_scores,
    importances,pvalues,n_features):
    if classification_label == 'classes':
        classification_type = 'Water/Ice'
    if classification_label == 'subclasses':
        classification_type = 'OW/RW/SI/FLI'

    fig, axes = plt.subplots(1,2,figsize=(30,10))
    plt.suptitle('GLCM{} Feature Selection'.format(str(w)),fontweight='bold')

    ymin = 0.99*np.min(rfecv_scores)
    ymax = 1.01*np.max(rfecv_scores)
    subplotname = 'RFECV Accuracies & Associated p-values'# and T-test p-values per chosen
        number of features'

    if len(rfecv_features) < 18:
```

```
        xbound = 0.98
    else:
        xbound = len(rfecv_features)/36*0.95
    ybound = 0.2

    axes[0].title.set_text(subplotname)
    axes[0].title.set_fontweight('bold')
    axes[0].set_xlabel('Number of selected features', labelpad=20, color='black')
    axes[0].set_ylabel('Accuracy [-]', labelpad=20, color='black')
    axes[0].plot(range(1,len(feature_names)+1), rfecv_scores, color='black', linewidth=3)
    optimal_line = axes[0].vlines(len(rfecv_features),ymin,ymax,colors='black',linestyles='
        dashed')
    chosen_line = axes[0].vlines(n_features,ymin,ymax,colors='tab:blue',linestyles='dashed')
    axes[0].set_ylim(ymin,ymax)
    axes[0].set_xlim(1,len(feature_names))
    axes[0].grid(axis='x')
    axes[0].legend([optimal_line,chosen_line],['Optimal','Chosen'],loc='best',bbox_to_anchor
        =(xbound,ybound))

    axesTwin = axes[0].twinx()
    axesTwin.set_ylabel('p-value [-]', labelpad=20, color='black')
    axesTwin.plot(range(1, len(feature_names) + 1), pvalues, color='red', linewidth=3)
    axesTwin.tick_params(axis='y', labelcolor='red', labelsize=25)
    axesTwin.set_ylim(0,1)

    sorted_indeces = importances.argsort()
    features_sorted = [rfecv_features[index] for index in sorted_indeces]
    importances_sorted = np.sort(importances)
    importancename = 'Feature Importances for n={}'.format(len(rfecv_features))

    axes[1].title.set_text(importancename)
    axes[1].title.set_fontweight('bold')
    axes[1].xaxis.grid(True)
    axes[1].barh(features_sorted,importances_sorted)
    chosen = axes[1].hlines(len(features_sorted)-n_features-0.5,0,1,colors='black',linestyles
        ='dashed')
    axes[1].set_xlim(0,importances_sorted[-1]*1.01)
    axes[1].set_ylabel('Feature Name', labelpad=20, fontsize=30)
    axes[1].set_xlabel('Feature Importance Score [-]',fontsize=30)

    plt.tight_layout()

    return fig

## (5.2) GLCM5
features_glcm5, accuracies_glcm5 = RFE_CV(rf,5,feature_names,'classes',visualize=False)
importances_glcm5 = FeatureImportanceScores(rf,5,feature_names,features_glcm5,'classes',
    visualize=False)
pvalues_glcm5 = StatisticalSignificance(rf,5,feature_names,'classes',visualize=False)

n = 6
featureselection_glcm5 = FeatureSelectionFigure(5,'classes',property_names,features_glcm5,
                                                   accuracies_glcm5,importances_glcm5,
                                                       pvalues_glcm5,n)

sorted_indeces = importances_glcm5.argsort()
features_sorted = [features_glcm5[index] for index in sorted_indeces][::-1][:n]
correlations = heavilyCorrelated(sample_data_glcm5[0],property_names,0.9)
corr = []
for i in range(len(features_sorted)):
    index = np.argwhere(np.array(feature_names) == features_sorted[i])[0][0]
    print(features_sorted[i]+': '+str(correlations[0][index]))
    print(features_sorted[i]+': '+str(correlations[1][index]))

## (5.3) GLCM11
features_glcm11, accuracies_glcm11 = RFE_CV(rf,11,feature_names,'classes',visualize=False)
importances_glcm11 = FeatureImportanceScores(rf,11,feature_names,features_glcm11,'classes',
    visualize=False)
pvalues_glcm11 = StatisticalSignificance(rf,11,feature_names,'classes',visualize=False)

n = 6
```

```
featureselection_glcm11 = FeatureSelectionFigure(11,'classes',property_names,features_glcm11,
                                                  accuracies_glcm11,importances_glcm11,
                                                      pvalues_glcm11,n)

sorted_indeces = importances_glcm11.argsort()
features_sorted = [features_glcm11[index] for index in sorted_indeces][::-1][:n]

correlations = heavilyCorrelated(sample_data_glcm11[0],property_names,0.9)
corr = []
for i in range(len(features_sorted)):
    index = np.argwhere(np.array(feature_names) == features_sorted[i])[0][0]
    print(features_sorted[i]+': '+str(correlations[0][index]))
    print(features_sorted[i]+': '+str(correlations[1][index]))

## (5.4) GLCM21
features_glcm21, accuracies_glcm21 = RFE_CV(rf,21,feature_names,'classes',visualize=False)
importances_glcm21 = FeatureImportanceScores(rf,21,feature_names,features_glcm21,'classes',
    visualize=False)
pvalues_glcm21 = StatisticalSignificance(rf,21,feature_names,'classes',visualize=False)

n = 5
featureselection_glcm21 = FeatureSelectionFigure(21,'classes',property_names,features_glcm21,
                                                  accuracies_glcm21,importances_glcm21,
                                                      pvalues_glcm21,n)

sorted_indeces = importances_glcm21.argsort()
features_sorted = [features_glcm21[index] for index in sorted_indeces][::-1][:n]

correlations = heavilyCorrelated(sample_data_glcm21[0],property_names,0.9)
corr = []
for i in range(len(features_sorted)):
    index = np.argwhere(np.array(feature_names) == features_sorted[i])[0][0]
    print(features_sorted[i]+': '+str(correlations[0][index]))
    print(features_sorted[i]+': '+str(correlations[1][index]))
```

B.7. Visualization of Feature Data

7_DataVisualization

```
#%% (1) Code set-up
## (1.1) Import packages
import ee
import geemap
import numpy as np
import os
import matplotlib.pyplot as plt

## (1.2) Initialize
ee.Initialize()

## (1.3) Parameterization & Data Import
properties = ['HH','HV','HH_savg','HV_savg','HV_dvar']
classes = ['Water','Ice']
subclasses = ['Open Water','Rough Water','Sea Ice','Floating Land Ice']
colors = ['C0','white','C0','C0','White','White']
fontsize = 25

TV11 = ee.FeatureCollection('users/skjeltmaps/TrainingValues_w11_1000pixels')  .map(lambda
    feat: feat.set('HH_contrast',feat.get('HH_contras')).set('HV_contrast',feat.get('
    HV_contras')))
TV5 = ee.FeatureCollection('users/skjeltmaps/TrainingValues_w5_1000pixels')  .map(lambda feat
    : feat.set('HH_contrast',feat.get('HH_contras')).set('HV_contrast',feat.get('HV_contras')
    ))
TV21 = ee.FeatureCollection('users/skjeltmaps/TrainingValues_w21_1000pixels')  .map(lambda
    feat: feat.set('HH_contrast',feat.get('HH_contras')).set('HV_contrast',feat.get('
    HV_contras')))

#%% (2) Boxplots of Data Sets
## (2.1) Functions
def makeBoxPlot(subplot,data,propertyname,ylims,fontsize):
```

```
    plt.rcParams.update({’axes.facecolor’:’lightgrey’})
    plt.subplot(subplot[0],subplot[1],subplot[2])

    bp = plt.boxplot(data, notch=0, sym=’+’, vert=1, whis=1.5, patch_artist=True,
                     boxprops={”edgecolor”: ”white”,”linewidth”: 0.5})
    for patch, color in zip(bp[’boxes’], colors):
        patch.set_facecolor(color)
    for median in bp[’medians’]:
        median.set_color(’black’)

    xname = ’’
    plt.xlabel(xname,fontsize=fontsize)
    yname = propertyname
    plt.ylabel(yname,fontsize=fontsize)

    plt.xticks(np.arange(1,len(data)+1),[’Water’,’Ice’,’Open\nWater’,’Rough\nWater’,’Sea\nIce’,’Floating\nLand Ice’])
    plt.tick_params(labelsize=fontsize,labelrotation=0)

    plt.vlines(2.5,ylims[0],ylims[1],’black’,’dashed’)
    plt.ylim(ylims)
    plt.grid(axis=’y’,color=’black’,linewidth=0.5)

    return _

## (2.2) GLCM5
sampledata = [[],[],[],[],[]]

for i in range(len(properties)):
    prop = properties[i]
    for j,subname in enumerate(classes):
        sampledata[i].append(TV5.filterMetadata(’classes’,’equals’,j+1).aggregate_array(prop).getInfo())
    for k,name in enumerate(subclasses):
        sampledata[i].append(TV5.filterMetadata(’subclasses’,’equals’,k+1).aggregate_array(prop).getInfo())

plt.figure(figsize=(30,20))
_ = makeBoxPlot([2,2,1],sampledata[0],’Intensity (HH) [dB]’,[-55,5],fontsize)
_ = makeBoxPlot([2,2,3],sampledata[1],’Intensity (HV) [dB]’,[-60,0],fontsize)
_ = makeBoxPlot([3,2,2],sampledata[2],’Sum Average\n(HH) [-]’,[-85,5],fontsize)
_ = makeBoxPlot([3,2,4],sampledata[3],’Sum Average\n(HV) [-]’,[-105,-5],fontsize)
_ = makeBoxPlot([3,2,6],sampledata[4],’Difference Variance\n(HV) [-]’,[-2,23],fontsize)

plt.show()

## (2.3) GLCM11
sampledata = [[],[],[],[],[]]

for i in range(len(properties)):
    prop = properties[i]
    for j,subname in enumerate(classes):
        sampledata[i].append(TV11.filterMetadata(’classes’,’equals’,j+1).aggregate_array(prop).getInfo())
    for k,name in enumerate(subclasses):
        sampledata[i].append(TV11.filterMetadata(’subclasses’,’equals’,k+1).aggregate_array(prop).getInfo())

plt.figure(figsize=(30,20))
_ = makeBoxPlot([2,2,1],sampledata[0],’Intensity (HH) [dB]’,[-55,5],fontsize)
_ = makeBoxPlot([2,2,3],sampledata[1],’Intensity (HV) [dB]’,[-60,0],fontsize)
_ = makeBoxPlot([3,2,2],sampledata[2],’Sum Average\n(HH) [-]’,[-85,5],fontsize)
_ = makeBoxPlot([3,2,4],sampledata[3],’Sum Average\n(HV) [-]’,[-105,-5],fontsize)
_ = makeBoxPlot([3,2,6],sampledata[4],’Difference Variance\n(HV) [-]’,[-2,23],fontsize)

plt.show()

## (2.4) GLCM21
sampledata = [[],[],[],[],[]]

for i in range(len(properties)):
```

```
    prop = properties[i]
    for j,subname in enumerate(classes):
        sampledata[i].append(TV21.filterMetadata(’classes’,’equals’,j+1).aggregate_array(prop).getInfo())
    for k,name in enumerate(subclasses):
        sampledata[i].append(TV21.filterMetadata(’subclasses’,’equals’,k+1).aggregate_array(prop).getInfo())

plt.figure(figsize=(30,20))
_ = makeBoxPlot([2,2,1],sampledata[0],’Intensity (HH) [dB]’,[-55,5],fontsize)
_ = makeBoxPlot([2,2,3],sampledata[1],’Intensity (HV) [dB]’,[-60,0],fontsize)
_ = makeBoxPlot([3,2,2],sampledata[2],’Sum Average\n(HH) [-]’,[-85,5],fontsize)
_ = makeBoxPlot([3,2,4],sampledata[3],’Sum Average\n(HV) [-]’,[-105,-5],fontsize)
_ = makeBoxPlot([3,2,6],sampledata[4],’Difference Variance\n(HV) [-]’,[-2,23],fontsize)

plt.show()
```

B.8. Classification & Accuracy Assessment

8_Classification

```
#%% (1) Code set-up
## (1.1) Import packages
import ee
import geemap
import numpy as np
import os
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

## (1.2) Initialize
ee.Initialize()

## (1.3) Parameterization & Data Import
classifier = ee.Classifier.smileRandomForest(
    numberOfTrees = 10,
    variablesPerSplit = 2,
    maxNodes = None,
    minLeafPopulation = 1,
    seed = 1)

plt.style.use(”seaborn”)
SMALL_SIZE = 18
MEDIUM_SIZE = 20
BIGGER_SIZE = 22

plt.rc(’font’, size=SMALL_SIZE)          # controls default text sizes
plt.rc(’axes’, titlesize=SMALL_SIZE)     # fontsize of the axes title
plt.rc(’axes’, labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc(’xtick’, labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc(’ytick’, labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc(’legend’, fontsize=SMALL_SIZE)    # legend fontsize
plt.rc(’figure’, titlesize=BIGGER_SIZE)  # fontsize of the figure title

TV11 = ee.FeatureCollection(’users/skjeltmaps/TrainingValues_w11_1000pixels’)  .map(lambda feat: feat.set(’HH_contrast’,feat.get(’HH_contras’)).set(’HV_contrast’,feat.get(’HV_contras’)))
TV5 = ee.FeatureCollection(’users/skjeltmaps/TrainingValues_w5_1000pixels’)  .map(lambda feat: feat.set(’HH_contrast’,feat.get(’HH_contras’)).set(’HV_contrast’,feat.get(’HV_contras’)))
TV21 = ee.FeatureCollection(’users/skjeltmaps/TrainingValues_w21_1000pixels’)  .map(lambda feat: feat.set(’HH_contrast’,feat.get(’HH_contras’)).set(’HV_contrast’,feat.get(’HV_contras’)))
TV0 = TV11.select([’HH’,’HV’,’classes’,’subclasses’,’name’,’utc’,’type’,’image’])

VV11 = ee.FeatureCollection(’users/skjeltmaps/ValidationValues_w11_1000pixels’)  .map(lambda feat: feat.set(’HH_contrast’,feat.get(’HH_contras’)).set(’HV_contrast’,feat.get(’HV_contras’)))
```

```
VV5 = ee.FeatureCollection(’users/skjeltmaps/ValidationValues_w5_1000pixels’)  .map(lambda feat: feat.set(’HH_contrast’,feat.get(’HH_contras’)).set(’HV_contrast’,feat.get(’HV_contras’)))
VV21 = ee.FeatureCollection(’users/skjeltmaps/ValidationValues_w21_1000pixels’)  .map(lambda feat: feat.set(’HH_contrast’,feat.get(’HH_contras’)).set(’HV_contrast’,feat.get(’HV_contras’)))
VV0 = VV11.select([’HH’,’HV’,’classes’,’subclasses’,’name’,’utc’,’type’,’image’])

#%% (2) Accuracy Assessment
## (2.1) BASE
TV = TV0
VV = VV0

n_features_training = TV.size().getInfo()
n_features_validation = VV.size().getInfo()

classchoices = [’classes’,’subclasses’]
bandchoices = [[’HH’,’HV’]]
dataframes0 = []
accuracies0 = []
kappas0 = []

for i in range(len(bandchoices)):
    bands = bandchoices[i]
    print(’Band choice: ’,bands)

    for j in range(len(classchoices)):
        classchoice = classchoices[j]
        print(’Class choice: ’,classchoice)
        if classchoice == ’classes’:
            classnames = [’Water’,’Ice’]
        elif classchoice == ’subclasses’:
            classnames = [’Open Water’,’Rough Water’,’Sea Ice’,’Floating Land Ice’]

        trained = classifier.train(TV,classchoice,bands)
        validated = VV.classify(trained)

        confusion_matrix_validation = np.asarray(validated.errorMatrix(classchoice,’classification’).getInfo())[1:,1:]
        dataframes0.append(pd.DataFrame(confusion_matrix_validation/confusion_matrix_validation.sum(axis=1),columns=classnames,index=classnames))

        print(’Validation Data Results:’)
        print(dataframes0[j].round(3))
        accuracies0.append(validated.errorMatrix(classchoice,’classification’).accuracy().getInfo())
        kappas0.append(validated.errorMatrix(classchoice,’classification’).kappa().getInfo())
        print(’Overall accuracy: ’,np.round(accuracies0[j],4),’ (kappa: ’,np.round(kappas0[j],4),’)’)

## (2.2) GLCM5
TV = TV5
VV = VV5
print(’Window size: 5’)

n_features_training = TV.size().getInfo()
n_features_validation = VV.size().getInfo()

classchoices = [’classes’,’subclasses’]
bandchoices = [
    [’HH_savg’, ’HV_savg’, ’HV_dvar’]
]
dataframes5 = []
accuracies5 = []
kappas5 = []

for i in range(len(bandchoices)):
    bands = bandchoices[i]
    print(’Band choice: ’,bands)

    for j in range(len(classchoices)):
```

```
        classchoice = classchoices[j]
        print('Class choice: ',classchoice)
        if classchoice == 'classes':
            classnames = ['Water','Ice']
        elif classchoice == 'subclasses':
            classnames = ['Open Water','Rough Water','Sea Ice','Floating Land Ice']

        trained = classifier.train(TV,classchoice,bands)
        validated = VV.classify(trained)

        confusion_matrix_validation = np.asarray(validated.errorMatrix(classchoice,'classification').getInfo())[1:,1:]
        dataframes5.append(pd.DataFrame(confusion_matrix_validation/confusion_matrix_validation.sum(axis=1),columns=classnames,index=classnames))

        print('Validation Data Results:')
        print(dataframes5[j].round(3))
        accuracies5.append(validated.errorMatrix(classchoice,'classification').accuracy().getInfo())
        kappas5.append(validated.errorMatrix(classchoice,'classification').kappa().getInfo())
        print('Overall accuracy: ',np.round(accuracies5[j],4),' (kappa: ',np.round(kappas5[j],4),')')

## (2.3) GLCM11
TV = TV11
VV = VV11
print('Window size: 11')

n_features_training = TV.size().getInfo()
n_features_validation = VV.size().getInfo()

classchoices = ['classes','subclasses']
bandchoices = [
    ['HH_savg', 'HV_savg', 'HV_dvar']
]
dataframes11 = []
accuracies11 = []
kappas11 = []

for i in range(len(bandchoices)):
    bands = bandchoices[i]
    print('Band choice: ',bands)

    for j in range(len(classchoices)):
        classchoice = classchoices[j]
        print('Class choice: ',classchoice)
        if classchoice == 'classes':
            classnames = ['Water','Ice']
        elif classchoice == 'subclasses':
            classnames = ['Open Water','Rough Water','Sea Ice','Floating Land Ice']

        trained = classifier.train(TV,classchoice,bands)
        validated = VV.classify(trained)

        confusion_matrix_validation = np.asarray(validated.errorMatrix(classchoice,'classification').getInfo())[1:,1:]
        dataframes11.append(pd.DataFrame(confusion_matrix_validation/confusion_matrix_validation.sum(axis=1),columns=classnames,index=classnames))

        print('Validation Data Results:')
        print(dataframes11[j].round(3))
        accuracies11.append(validated.errorMatrix(classchoice,'classification').accuracy().getInfo())
        kappas11.append(validated.errorMatrix(classchoice,'classification').kappa().getInfo())
        print('Overall accuracy: ',np.round(accuracies11[j],4),' (kappa: ',np.round(kappas11[j],4),')')

## (2.3) GLCM21
TV = TV21
VV = VV21
```

```python
print('Window size: 21')

n_features_training = TV.size().getInfo()
n_features_validation = VV.size().getInfo()

classchoices = ['classes','subclasses']
bandchoices = [
    ['HH_savg', 'HV_savg', 'HV_dvar']
    ]
dataframes21 = []
accuracies21 = []
kappas21 = []

for i in range(len(bandchoices)):
    bands = bandchoices[i]
    print('Band choice: ',bands)

    for j in range(len(classchoices)):
        classchoice = classchoices[j]
        print('Class choice: ',classchoice)
        if classchoice == 'classes':
            classnames = ['Water','Ice']
        elif classchoice == 'subclasses':
            classnames = ['Open Water','Rough Water','Sea Ice','Floating Land Ice']

        trained = classifier.train(TV,classchoice,bands)
        validated = VV.classify(trained)

        confusion_matrix_validation = np.asarray(validated.errorMatrix(classchoice,'classification').getInfo())[1:,1:]
        dataframes21.append(pd.DataFrame(confusion_matrix_validation/confusion_matrix_validation.sum(axis=1),columns=classnames,index=classnames))

        print('Validation Data Results:')
        print(dataframes21[j].round(3))
        accuracies21.append(validated.errorMatrix(classchoice,'classification').accuracy().getInfo())
        kappas21.append(validated.errorMatrix(classchoice,'classification').kappa().getInfo())
        print('Overall accuracy: ',np.round(accuracies21[j],4),' (kappa: ',np.round(kappas21[j],4),')')

#%% (3) Visualization
## (3.1) Main Classes Classification (Water-Ice)
plt.style.use("seaborn")
SMALL_SIZE = 40
MEDIUM_SIZE = 50
BIGGER_SIZE = 60

plt.rc('font', size=SMALL_SIZE)          # controls default text sizes
plt.rc('axes', titlesize=SMALL_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=SMALL_SIZE)    # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE)  # fontsize of the figure title

plt.figure(figsize=(40,10))
plt.subplot(1,4,1)
heat_map = sns.heatmap(
    dataframes0[0].round(2),
    linewidth = 2 ,
    vmin = 0,
    vmax = 1,
    annot = True,
    cmap = plt.cm.Blues,
    cbar = False,
    cbar_kws = {'shrink': 0.8},
    square = True,
    xticklabels = ['Water','Ice'],
    yticklabels = ['Water','Ice'])
```

```
plt.yticks(rotation=0)
plt.title('BASE\nAccuracy = {} (kappa = {})'.format(np.round(accuracies0[0],2),np.round(
    kappas0[0],2)))

plt.subplot(1,4,2)
heat_map = sns.heatmap(
    dataframes5[0].round(2),
    linewidth = 2 ,
    vmin = 0,
    vmax = 1,
    annot = True,
    cmap = plt.cm.Blues,
    cbar = False,
    cbar_kws = {'shrink': 0.8},
    square = True,
    xticklabels = ['Water','Ice'],
    yticklabels = ['Water','Ice'])
plt.yticks(rotation=0)
plt.title('GLCM5\nAccuracy = {} (kappa = {})'.format(np.round(accuracies5[0],2),np.round(
    kappas5[0],2)))

plt.subplot(1,4,3)
heat_map = sns.heatmap(
    dataframes11[0].round(2),
    linewidth = 2 ,
    vmin = 0,
    vmax = 1,
    annot = True,
    cmap = plt.cm.Blues,
    cbar = False,
    cbar_kws = {'shrink': 0.8},
    square = True,
    xticklabels = ['Water','Ice'],
    yticklabels = ['Water','Ice'])
plt.yticks(rotation=0)
plt.title('GLCM11\nAccuracy = {} (kappa = {})'.format(np.round(accuracies11[0],2),np.round(
    kappas11[0],2)))

plt.subplot(1,4,4)
heat_map = sns.heatmap(
    dataframes21[0].round(2),
    linewidth = 2 ,
    vmin = 0,
    vmax = 1,
    annot = True,
    cmap = plt.cm.Blues,
    cbar_kws = {'shrink': 0.8},
    square = False,
    xticklabels = ['Water','Ice'],
    yticklabels = ['Water','Ice'])
plt.yticks(rotation=0)
plt.title('GLCM21\nAccuracy = {} (kappa = {})'.format(np.round(accuracies21[0],2),np.round(
    kappas21[0],2)))

plt.tight_layout()
plt.show()

## (3.2) Sub-class Classification (OW-RW-SI-FLI)
plt.style.use("seaborn")
SMALL_SIZE = 25
MEDIUM_SIZE = 30
BIGGER_SIZE = 35

plt.rc('font', size=SMALL_SIZE)          # controls default text sizes
plt.rc('axes', titlesize=SMALL_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=SMALL_SIZE)    # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE)  # fontsize of the figure title

```

```python
plt.figure(figsize=(20,20))
plt.subplot(2,2,1)
heat_map = sns.heatmap(
    dataframes0[1].round(2),
    linewidth = 2 ,
    vmin = 0,
    vmax = 1,
    annot = True,
    cmap = plt.cm.Blues,
    cbar_kws = {'shrink': 0.8},
    square = True,
    xticklabels = ['Open\nWater','Rough\nWater','Sea\nIce','Floating\nLand Ice'],
    yticklabels = ['Open\nWater','Rough\nWater','Sea\nIce','Floating\nLand Ice'])
plt.yticks(rotation=0)
plt.title('BASE\nAccuracy = {} (kappa = {})'.format(np.round(accuracies0[1],2),np.round(
    kappas0[1],2)))

plt.subplot(2,2,2)
heat_map = sns.heatmap(
    dataframes5[1].round(2),
    linewidth = 2 ,
    vmin = 0,
    vmax = 1,
    annot = True,
    cmap = plt.cm.Blues,
    cbar_kws = {'shrink': 0.8},
    square = True,
    xticklabels = ['Open\nWater','Rough\nWater','Sea\nIce','Floating\nLand Ice'],
    yticklabels = ['Open\nWater','Rough\nWater','Sea\nIce','Floating\nLand Ice'])
plt.yticks(rotation=0)
plt.title('GLCM5\nAccuracy = {} (kappa = {})'.format(np.round(accuracies5[1],2),np.round(
    kappas5[1],2)))

plt.subplot(2,2,3)
heat_map = sns.heatmap(
    dataframes11[1].round(2),
    linewidth = 2 ,
    vmin = 0,
    vmax = 1,
    annot = True,
    cmap = plt.cm.Blues,
    cbar_kws = {'shrink': 0.8},
    square = True,
    xticklabels = ['Open\nWater','Rough\nWater','Sea\nIce','Floating\nLand Ice'],
    yticklabels = ['Open\nWater','Rough\nWater','Sea\nIce','Floating\nLand Ice'])
plt.yticks(rotation=0)
plt.title('GLCM11\nAccuracy = {} (kappa = {})'.format(np.round(accuracies11[1],2),np.round(
    kappas11[1],2)))

plt.subplot(2,2,4)
heat_map = sns.heatmap(
    dataframes21[1].round(2),
    linewidth = 2 ,
    vmin = 0,
    vmax = 1,
    annot = True,
    cmap = plt.cm.Blues,
    cbar_kws = {'shrink': 0.8},
    square = True,
    xticklabels = ['Open\nWater','Rough\nWater','Sea\nIce','Floating\nLand Ice'],
    yticklabels = ['Open\nWater','Rough\nWater','Sea\nIce','Floating\nLand Ice'])
plt.yticks(rotation=0)
plt.title('GLCM21\nAccuracy = {} (kappa = {})'.format(np.round(accuracies21[1],2),np.round(
    kappas21[1],2)))

plt.tight_layout()
plt.show()
```

B.9. Export of Classification & Post-Processing Results

9_Visualizations

```
#%% (1) Code set-up
## (1.1) Import packages
import ee
import geemap
import numpy as np
import os

## (1.2) Initialize
ee.Initialize()

## (1.3) Parameterization & Data Import
DEM = ee.Image(’CPOM/CryoSat2/ANTARCTICA_DEM’).select(’elevation’)
IceShelves = ee.FeatureCollection(”users/sophiederoda/IceShelf_Antarctica”)
ROIs = ee.FeatureCollection(’users/skjeltmaps/ISCLSegments20km’)

AmundsenEmbayment = ee.Geometry.Polygon(
    coords = [[[-108.406159, -75.468542], [-108.406159, -71.238725], [-96.219667, -71.238725], [-96.219667, -75.468542], [-108.406159, -75.468542]]],
    geodesic = False,
    proj = ’EPSG:4326’)
CarneyCoast = ee.Geometry.Polygon(
    coords = [[[-146.815157, -74.904569], [-142.211175, -76.343611], [-128.293796, -74.877073], [-120.513741, -75.203774], [-120.513741, -73.617891], [-127.520609, -72.722358], [-146.815157, -74.904569]]],
    geodesic = False,
    proj = ’EPSG:4326’)
PineIslandGlacier = ee.Geometry({
    ’geodesic’: False,
    ’type’: ’Polygon’,
    ’coordinates’: [[[-105.449683, -75.564228], [-105.449683, -73.068832], [-98.639549, -73.068832], [-98.639549, -75.564228], [-105.449683, -75.564228]]]
})

trainValArea = AmundsenEmbayment.union(CarneyCoast,maxError=1e3)

## (1.3) Parameterization
start = ’2014-01-01’
end = ’2022-04-01’
crs = ’EPSG:3031’
S1band = ’HV’
dt = 4
elevation_threshold = 200

ReferenceData = ee.FeatureCollection(’users/skjeltmaps/RD_’+S1band+’_’+str(dt)+’hrs_’+start[:4]+start[5:7]+start[8:10]+’_’+end[:4]+end[5:7]+end[8:10])
S1OptMetadata = ee.FeatureCollection(’users/skjeltmaps/S1Opt_’+S1band+’_’+str(dt)+’hrs_Metadata_’+start[:4]+start[5:7]+start[8:10]+’_’+end[:4]+end[5:7]+end[8:10])

S1_UTC = ReferenceData.filterMetadata(’data’,’equals’,’S1’).aggregate_array(’utc’)
Opt_UTC = ReferenceData.filterMetadata(’data’,’not_equals’,’S1’).aggregate_array(’utc’)

S1name = ’S1’
HHmin = -35
HHmax = 5
HVmin = -40
HVmax = 0

S1_HH_params = {
’min’: HHmin,
’max’: HHmax}
S1_HV_params = {
’min’: HVmin,
’max’: HVmax}
S1_RGB_params = {
’min’: [HVmin,HHmin,HHmin],
’max’: [HVmax,HHmax,HHmax]}

```

```
L8name = ’L8’
L8bands = [’B4’,’B3’,’B2’]

S2name = ’S2’
S2bands = [’B4’, ’B3’, ’B2’]

RGBbands = [’Red’,’Green’,’Blue’]
RGB_params = {
  ’bands’: RGBbands,
  ’min’: 3000,
  ’max’: 15000}

TV11 = ee.FeatureCollection(’users/skjeltmaps/TrainingValues_w11_1000pixels’)   .map(lambda feat: feat.set(’HH_contrast’,feat.get(’HH_contras’)).set(’HV_contrast’,feat.get(’HV_contras’)))
TV5 = ee.FeatureCollection(’users/skjeltmaps/TrainingValues_w5_1000pixels’)   .map(lambda feat: feat.set(’HH_contrast’,feat.get(’HH_contras’)).set(’HV_contrast’,feat.get(’HV_contras’)))
TV21 = ee.FeatureCollection(’users/skjeltmaps/TrainingValues_w21_1000pixels’)   .map(lambda feat: feat.set(’HH_contrast’,feat.get(’HH_contras’)).set(’HV_contrast’,feat.get(’HV_contras’)))

VV11 = ee.FeatureCollection(’users/skjeltmaps/ValidationValues_w11_1000pixels’)   .map(lambda feat: feat.set(’HH_contrast’,feat.get(’HH_contras’)).set(’HV_contrast’,feat.get(’HV_contras’)))
VV5 = ee.FeatureCollection(’users/skjeltmaps/ValidationValues_w5_1000pixels’)   .map(lambda feat: feat.set(’HH_contrast’,feat.get(’HH_contras’)).set(’HV_contrast’,feat.get(’HV_contras’)))
VV21 = ee.FeatureCollection(’users/skjeltmaps/ValidationValues_w21_1000pixels’)   .map(lambda feat: feat.set(’HH_contrast’,feat.get(’HH_contras’)).set(’HV_contrast’,feat.get(’HV_contras’)))

## Edit visualization export folder is necessary
# html_dir = os.path.join(os.path.expanduser(’~’), ’Downloads’)
# if not os.path.exists(html_dir):
#     os.makedirs(html_dir)

#%% (2) Reconstruct Match Data Set
## (2.1) General Functions
def utcToLocal(image):
    centroid = ee.Image(image).geometry(maxError=1e3).centroid(maxError=1e3)
    longitude = centroid.coordinates().get(0)
    localTime = ee.Image(image).date().advance(ee.Number(longitude).divide(15).ceil().subtract(1),’hour’)
    localMillis = ee.Date(localTime).millis()

    return image
        .set(’utctime’,ee.Image(image).get(’system:time_start’))
        .set(’localtime’,localMillis)

def addID(image,name):
    id = ee.String(name+’_’).cat(ee.String(ee.Date(ee.Image(image).get(’localtime’)).format(’YYYY_MM_dd_KK:mma’)))
    return image.set(’id’,id).set(’data’,name)

def addRGBbands(image,bandnames):
    renamed = image.select(bandnames,RGBbands)
    rgb = image.addBands(renamed).select(RGBbands)
    return rgb

def addWB(image):
        wb = ee.List(ee.List(image.geometry(1e4).bounds(1e4).coordinates().get(0)).get(0)).get(0)
        return image.set(’WB’,wb)

def addMetadata(image,utcname,metadata):
        utc = ee.String(image.get(’system:time_start’))
        filtered = metadata.filter(ee.Filter.inList(utcname,[utc]))
        name = filtered.aggregate_array(’NAME’).distinct()
        segment = name.map(lambda name: filtered.filterMetadata(’NAME’,’equals’,name).aggregate_array(’SEG’))
```

```
        wb = ee.List(ee.List(image.geometry(1e4).bounds(1e4).coordinates().get(0)).get(0)).
            get(0)
        return image.set('NAME',name).set('SEG',segment).set('WB',wb)

def addMatchIndex(image,metadata):
    utc = image.get('system:time_start')
    match = metadata.filterMetadata('utc','equals',utc).first()
    matchIndex = ee.Number(match.get('match'))
    return image.set('match',matchIndex)

def angleNormalization(image):
    def degreesToRadians(object):
        return object.multiply(np.pi).divide(180)

    thetaRef = degreesToRadians(ee.Number(30))
    radians = degreesToRadians(image.select('angle'))

    angleCorr = image.select('angle').divide(thetaRef.cos().pow(ee.Number(2))).multiply(
        radians.cos().pow(ee.Number(2)))
    HH = image.select('HH').divide(thetaRef.cos().pow(ee.Number(2))).multiply(radians.cos().
        pow(ee.Number(2)))
    HV = image.select('HV').divide(thetaRef.cos().pow(ee.Number(2))).multiply(radians.cos().
        pow(ee.Number(2)))

    normalized = image.addBands(HH,overwrite=True).addBands(HV,overwrite=True).addBands(
        angleCorr)

    return normalized

def clipToMatchingImage(image,matchData):
    matchIndex = ee.Number(image.get('match'))
    matchCol = matchData.filterMetadata('match','equals',matchIndex)
    matchList = matchCol.toList(matchCol.size())
    GeomList = matchList.map(lambda image: ee.Image(image).geometry())

    def combineGeometries(geom2,geom1):
        geom = ee.Geometry(geom1).union(ee.Geometry(geom2),maxError=1e3)
        return geom

    matchGeometry = ee.Geometry(GeomList.iterate(
        function = combineGeometries,
        first = ee.Geometry(GeomList.get(0))
    ))
    clipped = image.clip(matchGeometry)

    return clipped

## (2.2) Reconstruction
S1 = ee.ImageCollection("COPERNICUS/S1_GRD")
    .filter(ee.Filter.inList('system:time_start',S1_UTC))
    .filter(ee.Filter.listContains('transmitterReceiverPolarisation',S1band))
    .map(utcToLocal)
    .map(angleNormalization)
    .map(lambda image:     image.updateMask(DEM.unmask().gt(elevation_threshold).Not()))
    .map(lambda image: addID(image,S1name))
    .map(lambda image: addMetadata(image,'S1_UTC',S1OptMetadata))
    .map(lambda image: addMatchIndex(image,ReferenceData))
    .sort('WB')

L8 = ee.ImageCollection("LANDSAT/LC08/C01/T2")
    .filter(ee.Filter.inList('system:time_start',Opt_UTC))
    .map(lambda image: addRGBbands(image,L8bands))
    .map(utcToLocal)
    .map(lambda image: addID(image,L8name))
    .map(lambda image: addMatchIndex(image,ReferenceData))
    .map(lambda image: addMetadata(image,'Opt_UTC',S1OptMetadata))
S2 = ee.ImageCollection("COPERNICUS/S2")
    .filter(ee.Filter.inList('system:time_start',Opt_UTC))
    .map(lambda image: addRGBbands(image,S2bands))
    .map(utcToLocal)
    .map(lambda image: addID(image,S2name))
```

```
    .map(lambda image: addMatchIndex(image,ReferenceData))
    .map(lambda image: addMetadata(image,'Opt_UTC',S1OptMetadata))
Opt = L8.merge(S2).sort('WB')
OptList = Opt.toList(Opt.size())

#%% (3) Create Training & Validation Data Set
## (3.1) Match functions
def matchDataSets(S1,Opt):
    def addMatchesAsBands(imgS1,Opt):
        matchIndex = ee.Number(imgS1.get('match'))
        OptCol = Opt.filterMetadata('match','equals',matchIndex)
        OptList = OptCol.toList(OptCol.size())
        GeomList = OptList.map(lambda image: ee.Image(image).geometry())

        def combineGeometries(geom2,geom1):
            geom = ee.Geometry(geom1).union(ee.Geometry(geom2),maxError=1e3)
            return geom

        OptGeometry = ee.Geometry(GeomList.iterate(
            function = combineGeometries,
            first = ee.Geometry(GeomList.get(0))
        ))
        S1Geometry = imgS1.geometry()

        OptImage = OptCol.median().clip(S1Geometry)
        S1Image = imgS1.clip(OptGeometry)
        matchAdded = S1Image.addBands(OptImage)

        return matchAdded

    matchesOfInterest = Opt.aggregate_array('match').distinct().sort()
    S1Filtered = S1.filter(ee.Filter.inList('match',matchesOfInterest))
    S1Opt = S1Filtered.map(lambda imgS1: addMatchesAsBands(imgS1,Opt))

    return S1Opt

## (3.2) Create Reference Data Set
Opt_Amundsen = Opt.filterBounds(AmundsenEmbayment)
Opt_Carney = Opt.filterBounds(CarneyCoast)

AmundsenMatches = Opt_Amundsen.aggregate_array('match').distinct().sort()
CarneyMatches = Opt_Carney.aggregate_array('match').distinct().sort()

S1_Amundsen = S1.filter(ee.Filter.inList('match',AmundsenMatches))
Opt_Amundsen = Opt.filter(ee.Filter.inList('match',AmundsenMatches))
S1Opt_Amundsen = matchDataSets(S1_Amundsen,Opt_Amundsen)

S1_Carney = S1.filter(ee.Filter.inList('match',CarneyMatches))
Opt_Carney = Opt.filter(ee.Filter.inList('match',CarneyMatches))
S1Opt_Carney = matchDataSets(S1_Carney,Opt_Carney)

trainIDsAmundsen = ee.List([1,2,5,6,7,8])
trainIDsCarney = ee.List([1,4,11])

S1Training = ee.ImageCollection(trainIDsAmundsen.map(lambda index: ee.Image(S1_Amundsen.
    toList(S1_Amundsen.size()).get(index))))
    .merge(ee.ImageCollection(trainIDsCarney.map(lambda index: ee.Image(S1_Carney.toList(
        S1_Carney.size()).get(index)))))
OptTraining = Opt.filter(ee.Filter.inList('match',S1Training.aggregate_array('match').
    distinct().sort()))
TrainingData = ee.ImageCollection(trainIDsAmundsen.map(lambda index: ee.Image(S1Opt_Amundsen.
    toList(S1Opt_Amundsen.size()).get(index))))
    .merge(ee.ImageCollection(trainIDsCarney.map(lambda index: ee.Image(S1Opt_Carney.toList(
        S1Opt_Carney.size()).get(index)))))

valIDsAmundsen = ee.List([0,3,4])
valIDsCarney = ee.List([10])

S1Validation = ee.ImageCollection(valIDsAmundsen.map(lambda index: ee.Image(S1_Amundsen.
    toList(S1_Amundsen.size()).get(index))))
    .merge(ee.ImageCollection(valIDsCarney.map(lambda index: ee.Image(S1_Carney.toList(
```

```
        S1_Carney.size()).get(index)))))
OptValidation = Opt.filter(ee.Filter.inList('match',S1Validation.aggregate_array('match').
    distinct().sort()))
ValidationData = ee.ImageCollection(valIDsAmundsen.map(lambda index: ee.Image(S1Opt_Amundsen.
    toList(S1Opt_Amundsen.size()).get(index))))
    .merge(ee.ImageCollection(valIDsCarney.map(lambda index: ee.Image(S1Opt_Carney.toList(
        S1Opt_Carney.size()).get(index)))))

## (2.3) Store Match Metadata
opttrnnames = OptTraining.aggregate_array('system:index')
s1trnnames = S1Training.aggregate_array('system:index')
optvalnames = OptValidation.aggregate_array('system:index')
s1valnames = S1Validation.aggregate_array('system:index')

# print(s1trnnames.getInfo())
# print(s1valnames.getInfo())
# print(opttrnnames.getInfo())
# print(optvalnames.getInfo())

opttrnutcs = OptTraining.aggregate_array('localtime').map(lambda utc: ee.Date(utc).format('
    YYYY/MM/dd HH:mm:ss'))
s1trnutcs = S1Training.aggregate_array('localtime').map(lambda utc: ee.Date(utc).format('YYYY
    /MM/dd HH:mm:ss'))
optvalutcs = OptValidation.aggregate_array('localtime').map(lambda utc: ee.Date(utc).format('
    YYYY/MM/dd HH:mm:ss'))
s1valutcs = S1Validation.aggregate_array('localtime').map(lambda utc: ee.Date(utc).format('
    YYYY/MM/dd HH:mm:ss'))

# print(s1trnutcs.getInfo())
# print(opttrnutcs.getInfo())

# print(s1valutcs.getInfo())
# print(optvalutcs.getInfo())

#%% (3) GLCM Features
## (3.1) Computation
windows = [5,11,21]

for i in range(len(windows)):
    w = windows[i]
    training_data = eval('TV'+str(w))

    features = ValidationData.select(['HH','HV'])
        .map(lambda image: image.addBands(image.select('HH').toInt32().glcmTexture(size=w,
            kernel=None)))
        .map(lambda image: image.addBands(image.select('HV').toInt32().glcmTexture(size=w,
            kernel=None)))
        .select(['HH','HV','HH_savg','HV_savg','HV_dvar'])

## (3.2) Export
## Leave commented!!
## Export was necessary to store image collections

# geemap.ee_export_image_collection_to_drive(
#     features,
#     descriptions=['Val1Features_w'+str(w),'Val2Features_w'+str(w),'Val3Features_w'+str(w),'
    Val4Features_w'+str(w)],
#     folder='GLCM',
#     scale=40,
#     crs=crs
# )

#%% (4) Classification
## (4.1) Computation
classifier = ee.Classifier.smileRandomForest(
    numberOfTrees = 10,
    variablesPerSplit = 2,
    maxNodes = None,
    minLeafPopulation = 1,
    seed = 1)
classchoice = 'classes'
```

```
BASE = ValidationData
    .select(['HH','HV'])
    .map(lambda image: image.classify(classifier.train(TV5,classchoice,ee.List(['HH','HV'])))
        .copyProperties(image))
    .map(lambda image: image.rename('BASE'))

GLCM5 = ValidationData
    .select(['HH','HV'])
    .map(lambda image: image.addBands(image.select('HH').toInt32().glcmTexture(size=5,kernel=
        None)))
    .map(lambda image: image.addBands(image.select('HV').toInt32().glcmTexture(size=5,kernel=
        None)))
    .select(['HH_savg','HV_savg','HV_dvar'])
    .map(lambda image: image.classify(classifier.train(TV5,classchoice,ee.List(['HH_savg','
        HV_savg','HV_dvar'])))
    .copyProperties(image))
    .map(lambda image: image.rename('GLCM5'))

GLCM11 = ValidationData
    .select(['HH','HV'])
    .map(lambda image: image.addBands(image.select('HH').toInt32().glcmTexture(size=11,kernel
        =None)))
    .map(lambda image: image.addBands(image.select('HV').toInt32().glcmTexture(size=11,kernel
        =None)))
    .select(['HH_savg','HV_savg','HV_dvar'])
    .map(lambda image: image.classify(classifier.train(TV11,classchoice,ee.List(['HH_savg','
        HV_savg','HV_dvar'])))
    .copyProperties(image))
    .map(lambda image: image.rename('GLCM11'))

GLCM21 = ValidationData
    .select(['HH','HV'])
    .map(lambda image: image.addBands(image.select('HH').toInt32().glcmTexture(size=21,kernel
        =None)))
    .map(lambda image: image.addBands(image.select('HV').toInt32().glcmTexture(size=21,kernel
        =None)))
    .select(['HH_savg','HV_savg','HV_dvar'])
    .map(lambda image: image.classify(classifier.train(TV21,classchoice,ee.List(['HH_savg','
        HV_savg','HV_dvar'])))
    .copyProperties(image))
    .map(lambda image: image.rename('GLCM21'))

## (4.2) Export
## Leave commented!!
## Export was necessary to create feature collections used in section below

# geemap.ee_export_image_collection_to_drive(
#     BASE,
#     descriptions=[
#         'Val1_BASE',
#         'Val2_BASE',
#         'Val3_BASE',
#         'Val4_BASE'],
#     folder='Classification',
#     scale=40,
#     crs=crs
# )

# geemap.ee_export_image_collection_to_drive(
#     GLCM5,
#     descriptions=[
#         'Val1_GLCM5',
#         'Val2_GLCM5',
#         'Val3_GLCM5',
#         'Val4_GLCM5'],
#     folder='Classification',
#     scale=40,
#     crs=crs
# )
```

```
# geemap.ee_export_image_collection_to_drive(
#     GLCM11,
#     descriptions=[
#         'Val1_GLCM11',
#         'Val2_GLCM11',
#         'Val3_GLCM11',
#         'Val4_GLCM11'],
#     folder='Classification',
#     scale=40,
#     crs=crs
# )

# geemap.ee_export_image_collection_to_drive(
#     GLCM21,
#     descriptions=[
#         'Val1_GLCM21',
#         'Val2_GLCM21',
#         'Val3_GLCM21',
#         'Val4_GLCM21'],
#     folder='Classification',
#     scale=40,
#     crs=crs
# )

#%% (5) Post-Processing: Area Filter
## (5.1) Connected Neighbours Computation
neighbours5 = GLCM5.map(lambda image: image.updateMask(image.eq(1)).connectedPixelCount(1024,
    False))
neighbours11 = GLCM11.map(lambda image: image.updateMask(image.eq(1)).connectedPixelCount
    (1024,False))
neighbours21 = GLCM21.map(lambda image: image.updateMask(image.eq(1)).connectedPixelCount
    (1024,False))

## (5.2) Export
## Leave commented!!
## Export was necessary to create feature collections used in section below

# geemap.ee_export_image_collection_to_drive(
#     neighbours5,
#     descriptions=[
#         'Val1_NN5',
#         'Val2_NN5',
#         'Val3_NN5',
#         'Val4_NN5'],
#     folder='ConnectedNeighbors',
#     scale=40,
#     crs=crs
# )

# geemap.ee_export_image_collection_to_drive(
#     neighbours11,
#     descriptions=[
#         'Val1_NN11',
#         'Val2_NN11',
#         'Val3_NN11',
#         'Val4_NN11'],
#     folder='ConnectedNeighbors',
#     scale=40,
#     crs=crs
# )

# geemap.ee_export_image_collection_to_drive(
#     neighbours21,
#     descriptions=[
#         'Val1_NN21',
#         'Val2_NN21',
#         'Val3_NN21',
#         'Val4_NN21'],
#     folder='ConnectedNeighbors',
#     scale=40,
#     crs=crs
```

```
# )

#%% (6) Export Optical Images
# geemap.ee_export_image_collection_to_drive(
#     OptValidation.select(RGBbands),
#     descriptions=['ValOptFull_1','ValOptFull_2','ValOptFull_3','ValOptFull_4'],
#     folder='export',
#     scale=Opt.first().select(0).projection().nominalScale(),
#     crs=crs
# )
```

Bibliography

Alley, K. E., Scambos, T. A., Alley, R. B., and Holschuh, N. (2019). Troughs developed in ice-stream shear margins precondition ice shelves for ocean-driven breakup. Added in review.

Alley, K. E., Scambos, T. A., Siegfried, M. R., and Fricker, H. A. (2016). Impacts of warm water on antarctic ice shelf stability through basal channel formation. *Nature Geoscience 2016 9:4*, 9:290–293. Added in review.

Barber, D. G. and Massom, R. A. (2007). Chapter 1 the role of sea ice in arctic and antarctic polynyas.

Bindschadler, R., Vaughan, D. G., and Vornberger, P. (2011). Variability of basal melt beneath the pine island glacier ice shelf, west antarctica. *Journal of Glaciology*, 57:581–595. Added in review.

Brocq, A. M. L., Ross, N., Griggs, J. A., Bingham, R. G., Corr, H. F. J., Ferraccioli, F., Jenkins, A., Jordan, T. A., Payne, A. J., Rippin, D. M., and Siegert, M. J. (2013). Evidence from ice shelves for channelized meltwater flow beneath the antarctic ice sheet. *Nature Geoscience 2013 6:11*, 6:945–948. Added in review.

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20:37–46.

Conners, R. W., Trivedi, M. M., and Harlow, C. A. (1984). Segmentation of a high-resolution urban scene using texture operators. *Computer Vision, Graphics, and Image Processing*, 25:273–310.

de Roda Husman, S., van der Sanden, J. J., Lhermitte, S., and Eleveld, M. A. (2021). Integrating intensity and context for improved supervised river ice classification from dual-pol sentinel-1 sar data. *International Journal of Applied Earth Observation and Geoinformation*, 101:102359.

Dierking, W. (2010). Mapping of different sea ice regimes using images from sentinel-1 and alos synthetic aperture radar. *IEEE Transactions on Geoscience and Remote Sensing*, 48:1045–1058. Added in review.

Dierking, W. (2013). Sea ice monitoring by synthetic aperture radar. *Oceanography*, 26. Added in review.

Dutrieux, P., Stewart, C., Jenkins, A., Nicholls, K. W., Corr, H. F. J., Rignot, E., and Steffen, K. (2014). Basal terraces on melting ice shelves. *Geophysical Research Letters*, 41:5506–5513. Added in review.

ESA (2012). *Sentinel-1 : ESA's radar observatory mission for GMES operational services*. ESA Communications.

Freeman, A. and Durden, S. L. (1998). A three-component scattering model for polarimetric sar data. *IEEE Transactions on Geoscience and Remote Sensing*, 36:963–973.

Gladish, C. V., Holland, D. M., Holland, P. R., and Price, S. F. (2012). Ice-shelf basal channels in a coupled ice/ocean model. *Journal of Glaciology*, 58:1227–1244.

Gorelick, N., Hancher, M., Dixon, M., Ilyushchenko, S., Thau, D., and Moore, R. (2017). Google earth engine: Planetary-scale geospatial analysis for everyone. *Remote Sensing of Environment*, 202:18–27. Added in review.

Gourmelen, N., Goldberg, D. N., Snow, K., Henley, S. F., Bingham, R. G., Kimura, S., Hogg, A. E., Shepherd, A., Mouginot, J., Lenaerts, J. T. M., Ligtenberg, S. R. M., and van de Berg, W. J. (2017). Channelized melting drives thinning under a rapidly melting antarctic ice shelf. *Geophysical Research Letters*, 44:9796–9804. Added in review.

Haralick, R. M., Dinstein, I., and Shanmugam, K. (1973). Textural features for image classification. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-3:610–621.

Hellmer, H. H., Kauker, F., Timmermann, R., Determann, J., and Rae, J. (2012). Twenty-first-century warming of a large antarctic ice-shelf cavity by a redirected coastal current. Added in review.

Hoekstra, M., Jiang, M., Clausi, D. A., and Duguay, C. (2020). Lake ice-water classification of radarsat-2 images by integrating irgs segmentation with pixel-based random forest labeling. *Remote Sensing*, 12:1425.

Hollands, T. and Dierking, W. (2016). Dynamics of the terra nova bay polynya: The potential of multi-sensor satellite observations. *Remote Sensing of Environment*, 187:30–48. Added in review.

IPCC (2014). *Climate change 2014: Synthesis Report*.

Jourdain, N. C., Mathiot, P., Merino, N., Durand, G., Sommer, J. L., Spence, P., Dutrieux, P., and Madec, G. (2017). Ocean circulation and sea-ice thinning induced by melting ice shelves in the amundsen sea. *Journal of Geophysical Research: Oceans*, 122:2550–2573.

Karvonen, J. and Hallikainen, M. (2009). Sea ice sar classification based on edge features. *International Geoscience and Remote Sensing Symposium (IGARSS)*, 3.

Karvonen, J., Similä, M., and Mäkynen, M. (2005). Open water detection from baltic sea ice radarsat-1 sar imagery. *IEEE Geoscience and Remote Sensing Letters*, 2:275–279.

Lazeroms, W. M., Jenkins, A., Gudmundsson, G. H., and Wal, R. S. V. D. (2018). Modelling present-day basal melt rates for antarctic ice shelves using a parametrization of buoyant meltwater plumes. *Cryosphere*, 12:49–70. Added to review.

Lhermitte, S., Sun, S., Shuman, C., Wouters, B., Pattyn, F., Wuite, J., Berthier, E., and Nagler, T. (2020). Damage accelerates ice shelf instability and mass loss in amundsen sea embayment. *Proceedings of the National Academy of Sciences of the United States of America*, 117:24735–24741.

Lohse, J., Doulgeris, A. P., and Dierking, W. (2021). Incident angle dependence of sentinel-1 texture features for sea ice classification. *Remote Sensing 2021, Vol. 13, Page 552*, 13:552.

Long, D. G., Collyer, R. S., Reed, R., and Arnold, D. V. (1996). Dependence of the normalized radar cross section of water waves on bragg wavelength-wind speed sensitivity. *IEEE Transactions on Geoscience and Remote Sensing*, 34:656–666.

Mankoff, K. D., Jacobs, S. S., Tulaczyk, S. M., and Stammerjohn, S. E. (2012). The role of pine island glacier ice shelf basal channels in deep-water upwelling, polynyas and ocean circulation in pine island bay, antarctica. *Annals of Glaciology*, 53:123–128. Added in review.

Maqueda, M. A. M., Willmott, A. J., and Biggs, N. R. T. (2004). Polynya dynamics: a review of observations and modeling. *Reviews of Geophysics*, 42. Added in review.

Mermoz, S., Allain, S., Bernier, M., Pottier, E., and Gherboudj, I. (2009). Classification of river ice using polarimetric sar data. *Canadian Journal of Remote Sensing*, 35:460–473.

Pattyn, F. and Morlighem, M. (2020). The uncertain future of the antarctic ice sheet. *Science*, 367:1331–1335.

Rignot, E. and Steffen, K. (2008). Channelized bottom melting and stability of floating ice shelves. *Geophysical Research Letters*, 35.

Sergienko, O. V. (2013). Basal channels on ice shelves. *Journal of Geophysical Research: Earth Surface*, 118:1342–1355. Added in review.

Shean, D. E., Joughin, I. R., Dutrieux, P., Smith, B. E., and Berthier, E. (2019). Ice shelf basal melt rates from a high-resolution digital elevation model (dem) record for pine island glacier, antarctica. *Cryosphere*, 13:2633–2656. Added in review.

Shelestov, A., Lavreniuk, M., Kussul, N., Novikov, A., and Skakun, S. (2017). Exploring google earth engine platform for big data processing: Classification of multi-temporal satellite imagery for crop mapping. *Frontiers in Earth Science*, 0:17. Added in review.

Stewart, C. L., Christoffersen, P., Nicholls, K. W., Williams, M. J. M., and Dowdeswell, J. A. (2019). Basal melting of ross ice shelf from solar heat absorption in an ice-front polynya. *Nature Geoscience 2019 12:6*, 12:435–440.

Topouzelis, K. and Singha, S. (2016). Incidence angle normalization of wide swath sar data for oceanographic applications. *Open Geosciences*, 8:450–464.

Williams, W. J., Carmack, E. C., and Ingram, R. G. (2007). Chapter 2 physical oceanography of polynyas.

Xu, L. and Li, J. (2015). Mapping sea ice from satellite sar imagery. pages 113–135. Added in review.