How to detect snow in SAR images?

Detecting snow in Synthetic Aperture Radar (SAR) images using machine learning involves several steps.

1. **Data collection:** Obtain a dataset of SAR images labeled with the presence or absence of snow. Ensure that the dataset is diverse and representative of different conditions.
2. **Data preprocessing:** SAR images often require specific preprocessing due to their nature. Preprocess the images to remove noise, normalize intensities, and handle speckle. Consider using SAR-specific libraries like pyroSAR or snappy for preprocessing.
   1. This example assumes that your SAR image is in a zip archive. Adjust the paths, filenames, and parameters according to your specific case.

import pyroSAR

from pyroSAR import identify

from pyroSAR.ancillary import groupby\_helper

def preprocess\_sar\_image(input\_path, output\_path):

# Identify the SAR product

product = identify(input\_path)

# Apply radiometric calibration (if needed - If the SAR image is not already in gamma nought (γ0) format, it is often necessary to perform radiometric calibration.)

if 'gamma0' not in product.polarizations:

product = product.resample(30).calibrate(1)

# Apply speckle filtering (SAR images are often affected by speckle noise, which can be reduced using speckle filtering techniques. In this example, a Lee filter is applied.)

product = product.filter(1, 'Lee')

# Extract metadata for geocoding (Geocoding involves projecting the SAR image onto a geographical coordinate system. This step requires additional data, such as a Digital Elevation Model (DEM). The example uses SRTM data for geocoding.)

metadata = groupby\_helper(product, 'outname', 'orbitnumber', 'date', 'platform', 'center\_lon', 'center\_lat', 'acquisition\_mode', 'polarizations', 'looks')

# Geocode the image (requires SRTM data, adjust the path accordingly)

product.geocode(target=output\_path, trgtsize=0.0001, overwrite=True, removeS1border=True, externalDEMFile='path/to/SRTM1/DEM/SRTM1.vrt', extern\_mask=True, rtc=True)

# Example usage

input\_sar\_image = 'path/to/your/sar/image.zip'

output\_preprocessed\_image = 'path/to/your/preprocessed/image.tif'

preprocess\_sar\_image(input\_sar\_image, output\_preprocessed\_image)

1. **Feature extraction:** Extract relevant features from the SAR images. These features might include texture, intensity, and statistical measures. You can use libraries like scikit-image or rasterio for feature extraction.
   1. **Backsatter intensity feature extraction:** These statistical features can be used as input features for machine learning models or combined with other features for more comprehensive analysis. Adjust the calculations or include additional features based on your specific requirements and the characteristics of your SAR images.

import numpy as np

import matplotlib.pyplot as plt

from skimage import io, color

def backscatter\_feature\_extraction(image\_path):

# Read the SAR image

sar\_image = io.imread(image\_path)

# Convert to grayscale

sar\_gray = color.rgb2gray(sar\_image)

# Calculate statistical features from the backscatter intensity (Compute statistical features from the backscatter intensity, such as mean, standard deviation, maximum, and minimum intensity values.)

mean\_intensity = np.mean(sar\_gray)

std\_intensity = np.std(sar\_gray)

max\_intensity = np.max(sar\_gray)

min\_intensity = np.min(sar\_gray)

# Display the original SAR image

plt.subplot(121)

plt.imshow(sar\_image, cmap='gray')

plt.title('Original SAR Image')

# Display the backscatter intensity distribution

plt.subplot(122)

plt.hist(sar\_gray.flatten(), bins=256, range=[0, 1], density=True, cumulative=False, color='gray', alpha=0.75)

plt.title('Backscatter Intensity Distribution')

plt.xlabel('Pixel Intensity')

plt.ylabel('Frequency')

plt.show()

print(f"Mean Intensity: {mean\_intensity}")

print(f"Standard Deviation Intensity: {std\_intensity}")

print(f"Maximum Intensity: {max\_intensity}")

print(f"Minimum Intensity: {min\_intensity}")

# Example usage

image\_path = 'path/to/your/preprocessed/image.tif'

backscatter\_feature\_extraction(image\_path)

* 1. **Texture analysis feature extraction:** Adjust the parameters (distance, angles, levels) based on your specific requirements and characteristics of the SAR images. Texture analysis can be combined with other feature extraction methods to improve the performance of your machine learning model for snow detection.

import numpy as np

import matplotlib.pyplot as plt

from skimage import io, color, img\_as\_ubyte

from skimage.feature import greycomatrix

from skimage import exposure

def texture\_analysis(image\_path, distance=1, angles=[0, np.pi/4, np.pi/2, 3\*np.pi/4], levels=256):

# Read the SAR image

sar\_image = io.imread(image\_path)

# Convert to grayscale

sar\_gray = color.rgb2gray(sar\_image)

# Convert to unsigned byte for GLCM computation (Compute the Gray-Level Co-occurrence Matrix (GLCM) using the greycomatrix function from skimage.feature. This matrix describes the distribution of pixel pairs with certain intensity values and spatial relationships.)

sar\_gray = img\_as\_ubyte(sar\_gray)

# Apply texture analysis using GLCM

glcm = greycomatrix(sar\_gray, distances=[distance], angles=angles, levels=levels,

symmetric=True, normed=True)

# Calculate texture features (contrast, dissimilarity, homogeneity, energy) (Extract texture features such as contrast, dissimilarity, homogeneity, and energy from the GLCM.)

contrast = np.mean((glcm[:, :, 0, 0] \* (np.arange(levels) - np.arange(levels).mean())\*\*2))

dissimilarity = np.mean(glcm[:, :, 0, 1])

homogeneity = np.mean(glcm[:, :, 0, 2])

energy = np.mean(glcm[:, :, 0, 3])

# Display the original image

plt.subplot(121)

plt.imshow(sar\_image)

plt.title('Original SAR Image')

# Display the GLCM

plt.subplot(122)

plt.imshow(np.log1p(glcm[:, :, 0, 0]), cmap=plt.cm.gray, interpolation='nearest')

plt.title('GLCM (Contrast)')

plt.show()

print(f"Contrast: {contrast}")

print(f"Dissimilarity: {dissimilarity}")

print(f"Homogeneity: {homogeneity}")

print(f"Energy: {energy}")

# Example usage

image\_path = 'path/to/your/preprocessed/image.tif'

texture\_analysis(image\_path)

1. **Labeling:** Ensure that your dataset is labeled with the presence or absence of snow. This is crucial for supervised learning.
   1. Automatically labeling SAR images for the presence or absence of snow can be done using **Thresholding on Intensity:** SAR images of snow-covered areas often have distinct intensity characteristics. You can set a threshold on the pixel intensity to differentiate between snow and non-snow regions. This threshold can be determined manually or through histogram analysis.

Adjust the threshold value according to the characteristics of your SAR images. You may need to experiment and fine-tune these methods based on the specific features of your dataset. Keep in mind that automatic labeling methods might not be perfect, and it's essential to validate and refine the labels through manual inspection or additional feedback loops to improve accuracy.

import cv2

import numpy as np

def label\_snow\_by\_intensity(image\_path, threshold=100):

# Read the SAR image

sar\_image = cv2.imread(image\_path, cv2.IMREAD\_GRAYSCALE)

# Apply thresholding

snow\_mask = (sar\_image > threshold).astype(np.uint8)

return snow\_mask

# Example usage

image\_path = 'path/to/your/sar/image.png'

snow\_mask = label\_snow\_by\_intensity(image\_path, threshold=120)

# 'snow\_mask' now contains binary labels where 1 represents snow-covered areas

1. **Splitting the data set:** Split your dataset into training and testing sets. The training set is used to train your machine learning model, and the testing set is used to evaluate its performance on unseen data. This ensures that you can evaluate the model's performance on unseen data.
   1. Make sure to replace the placeholder values for X and y with your actual feature and label data.

The test\_size parameter controls the percentage of the data that will be allocated for testing (here, 20%). Adjust this parameter based on your specific needs.

After splitting the data, you can proceed with training your machine learning model using the X\_train and y\_train datasets. The X\_test and y\_test datasets are reserved for evaluating the model's performance.

from sklearn.model\_selection import train\_test\_split

# Assuming X contains your features and y contains your labels (0 or 1 for no snow and snow, respectively)

# Adjust X and y based on your actual dataset structure

# Example features (replace this with your actual feature data)

X = [[feature1\_value, feature2\_value, ...], [feature1\_value, feature2\_value, ...], ...]

# Example labels (replace this with your actual label data)

y = [0, 1, 0, 1, ...]

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 'test\_size' determines the proportion of the dataset used for testing (here, 20%)

# 'random\_state' ensures reproducibility, using the same seed will yield the same split each time

# Print the sizes of the training and testing sets

print(f"Training set size: {len(X\_train)} samples")

print(f"Testing set size: {len(X\_test)} samples")

1. **Selecting a machine learning model:** Choose a suitable machine learning algorithm for your task. Common choices for image classification include Support Vector Machines (SVM), Random Forests, or Convolutional Neural Networks (CNNs).
   1. One of the simplest and widely used machine learning algorithms is the Support Vector Machine (SVM).
   2. If you have a large dataset or complex relationships, you might also consider exploring more advanced algorithms such as Random Forests.
2. **Model training:** Train your chosen model using the training dataset. Feed the extracted features into the model along with their corresponding labels.
   1. **Example for the SVM:** Replace the placeholder values for X\_train, y\_train, X\_test, and y\_test with your actual training and testing data. In this example, we use a linear kernel for the SVM (kernel='linear'), which is a good starting point for many classification problems. Depending on your dataset, you might need to explore other kernel options and tune hyperparameters for better performance.

from sklearn import svm

from sklearn.metrics import accuracy\_score

# Assuming X\_train contains your training features and y\_train contains your training labels

# Assuming X\_test contains your testing features and y\_test contains your testing labels

# Example training features (replace this with your actual training feature data)

X\_train = [[feature1\_value, feature2\_value, ...], [feature1\_value, feature2\_value, ...], ...]

# Example training labels (replace this with your actual training label data)

y\_train = [0, 1, 0, 1, ...]

# Example testing features (replace this with your actual testing feature data)

X\_test = [[feature1\_value, feature2\_value, ...], [feature1\_value, feature2\_value, ...], ...]

# Example testing labels (replace this with your actual testing label data)

y\_test = [0, 1, 0, 1, ...]

# Create an SVM classifier

svm\_classifier = svm.SVC(kernel='linear')

# Train the classifier

svm\_classifier.fit(X\_train, y\_train)

# Make predictions on the test set

predictions = svm\_classifier.predict(X\_test)

# Evaluate the model

accuracy = accuracy\_score(y\_test, predictions)

print(f'Accuracy: {accuracy}')

1. **Model evaluation:** Evaluate the model's performance on the testing dataset. Common evaluation metrics for binary classification tasks include accuracy, precision, recall, and F1 score.
   1. Replace the placeholder values for predictions and y\_test with your actual predicted labels and true labels, respectively.

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

# Assuming predictions contain the predicted labels and y\_test contains the true labels

# Example predictions (replace this with your actual predictions)

predictions = [0, 1, 0, 1, ...]

# Example true labels (replace this with your actual true labels)

y\_test = [0, 1, 0, 1, ...]

# Calculate and print accuracy (Accuracy: The ratio of correctly predicted instances to the total instances.)

accuracy = accuracy\_score(y\_test, predictions)

print(f'Accuracy: {accuracy}')

# Display confusion matrix (A table showing the number of true positives, true negatives, false positives, and false negatives.)

conf\_matrix = confusion\_matrix(y\_test, predictions)

print('Confusion Matrix:')

print(conf\_matrix)

# Display classification report (Provides precision, recall, and F1-score for each class. Precision is the ratio of correctly predicted positive observations to the total predicted positives, recall is the ratio of correctly predicted positive observations to the all observations in the actual class, and F1-score is the weighted average of precision and recall.)

class\_report = classification\_report(y\_test, predictions)

print('Classification Report:')

print(class\_report)

1. **Tuning and optimization:** Fine-tune your model parameters to improve its performance. Consider techniques like cross-validation and hyperparameter tuning.
   1. **Grid search example:** is a common technique used to search through a predefined set of hyperparameter values and find the combination that yields the best performance.

Adjust the param\_grid values based on your specific requirements and the characteristics of your dataset. Grid search helps you find the optimal hyperparameters for your model, improving its overall performance.

* 1. The param\_grid dictionary defines the hyperparameters and their potential values that the grid search will explore.
  2. The GridSearchCV object performs a cross-validated grid search, trying out all possible combinations of hyperparameters.

from sklearn import svm, datasets

from sklearn.model\_selection import GridSearchCV, train\_test\_split

from sklearn.metrics import accuracy\_score

# Load a sample dataset for demonstration purposes (replace this with your actual dataset)

iris = datasets.load\_iris()

X, y = iris.data, iris.target

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Define the SVM classifier

svm\_classifier = svm.SVC()

# Define the hyperparameter grid to search

param\_grid = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf', 'poly'], 'gamma': ['scale', 'auto']}

# Create a grid search object with the SVM classifier and hyperparameter grid

grid\_search = GridSearchCV(svm\_classifier, param\_grid, cv=5, scoring='accuracy')

# Perform grid search on the training data

grid\_search.fit(X\_train, y\_train)

# Print the best hyperparameters found by the grid search (The best hyperparameters found by the grid search are printed, and the best model is used to make predictions on the test set.)

best\_params = grid\_search.best\_params\_

print(f'Best Hyperparameters: {best\_params}')

# Make predictions on the test set using the best model

best\_model = grid\_search.best\_estimator\_

predictions = best\_model.predict(X\_test)

# Evaluate the model

accuracy = accuracy\_score(y\_test, predictions)

print(f'Accuracy on Test Set: {accuracy}')

1. **Deployment:** Once satisfied with the model's performance, deploy it for snow detection in new SAR images. Involves using your trained machine learning model to make predictions on new, unseen data.
   1. Example on how to save the trained Support Vector Machine (SVM) model and later use it for making predictions on new data: Replace the training and testing data with your actual dataset and adapt the code according to the specific requirements of your deployment environment.

We train an SVM classifier on a sample dataset (Iris dataset in this case) 🡪 We save the trained model to a file using joblib.dump 🡪 Later, we load the saved model using joblib.load 🡪 We make predictions on new data using the loaded model.

from sklearn import svm

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

import joblib

# Load a sample dataset for demonstration purposes (replace this with your actual dataset)

iris = load\_iris()

X, y = iris.data, iris.target

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Train an SVM classifier (you can replace this with your own training code)

svm\_classifier = svm.SVC(kernel='linear')

svm\_classifier.fit(X\_train, y\_train)

# Save the trained model to a file using joblib

model\_filename = 'svm\_model.joblib'

joblib.dump(svm\_classifier, model\_filename)

# Later, when you want to make predictions on new data:

# Load the saved model

loaded\_model = joblib.load(model\_filename)

# Assuming you have new data in the variable 'new\_data'

new\_data = [[4.8, 3.0, 1.4, 0.3]] # Replace this with your actual new data

# Make predictions using the loaded model

predictions = loaded\_model.predict(new\_data)

# Display the predictions

print(f'Predictions for new data: {predictions}')

Methods for Feature extraction:

1. **Backscatter Intensity Analysis**

Backsatter: the reflection of radar signals back to the radar antenna from the Earth's surface or other targets.

* Snow-covered areas tend to exhibit higher backscatter intensity compared to other surfaces due to their relatively smooth and uniform texture.
* Analyzing the backscatter intensity can help identify regions with higher reflectivity, which may indicate the presence of snow.
* Quick and computationally efficient method for an initial assessment.
* Easier to implement in a machine learning model.
  + Utilizing machine learning algorithms, such as supervised or unsupervised classification, can help automatically identify snow-covered areas based on training datasets.
  + Algorithms examples: Random Forest, Support Vector Machines, or Convolutional Neural Networks (CNNs) for this purpose.
* It can be improved with polarimetric parameters, and possibly texture in the images, that are also easier to implement in the model.
  + Polarimetric SAR data captures information about the polarization states of radar waves, providing a richer set of information compared to traditional single-polarization SAR data. The basic polarimetric parameters are derived from the scattering matrix, which describes how the incoming and outgoing polarizations of radar waves interact with the Earth's surface. These polarimetric parameters offer valuable insights into the physical properties of the Earth's surface and help in discriminating between different types of terrain and targets.
  + Snow-covered areas often have a smoother and more homogeneous texture compared to other land cover types.

1. **Temporal analysis**

* Comparing multiple SAR images taken at different times.
* Good for changes in snow cover, especially in dynamic environments, like melting, accumulation, or other seasonal variations.