Image classification without the use of machine learning

Cloud and Noncloud Image Classifier

The cloud/non-cloud image dataset consists of 200 RGB color images in two categories: cloud and noncloud. There are equal numbers of each

We're trying to build a classifier that can accurately label these images as cloud/non-cloud, and that relies on finding distinguishing features between the two types of images!

Note: All images come from the AMOS dataset (Archive of Many Outdoor Scenes).

Import resources

Before we get started on the project code, import the libraries that we'll need.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
import os
import glob #for loading images from a directory
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import cv2
import random
import numpy as np
```

```
# Image data directories
image_dir_training = "/content/drive/MyDrive/images/training"
image_dir_test = "/content/drive/MyDrive/images/test/"
```

Step 1: Load the datasets and visualize

These first few lines of code will load the training cloud/noncloud images and store all of them in a variable, `IMAGE_LIST`. This list contains the images and their associated label (cloud or non cloud).

For example, the first image-label pair in `IMAGE_LIST` can be accessed by index: `IMAGE_LIST[0][:]`.

```
def load_dataset(image_dir):
    '''This function loads in images and their labels and places them
in a list
    im_list[0][:] will be the first image-label pair in the list'''

    im_list = []
    image_types = ["cloud", "noncloud"]
```

```
    # Iterate through each color folder
    for im_type in image_types:

        # Iterate through each image file in each image_type folder
        # glob reads in any image with the extension
"image_dir/im_type/*"
        for file in glob.glob(os.path.join(image_dir, im_type, "*")):

            # Read in the image
            im = mpimg.imread(file)

            # Check if the image exists/if it's been correctly read-in
            if not im is None:
                # Append the image, and it's type (red, green, yellow)
to the image list
                im_list.append((im, im_type))

    return im_list

# Load training data
IMAGE_LIST = load_dataset(image_dir_training)
```

Step 2: Preprocess the data input images.

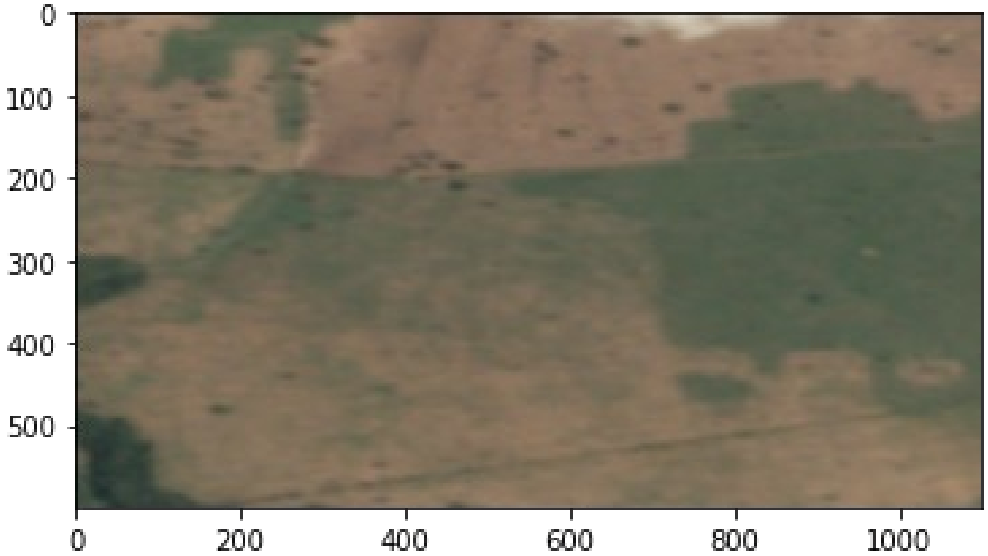
This function takes in a list of image-label pairs and outputs a **standardized** list of resized images and numerical labels.

1. Resizing every image to a standard size
2. Encode the target variables

```
def standardize_input(image):

    # Resize image and pre-process so that all "standard" images are
the same size
    standard_im = cv2.resize(image, (1100, 600))

    return standard_im

def encode(label):
    # encode cloud as 1, noncloud as 0
    numerical_val = 0
    if(label == 'cloud'):
        numerical_val = 1

    return numerical_val

def preprocess(image_list):

    #standardize and encode the input data
    standard_list = []

    # Iterate through all the image-label pairs
```

```
    for item in image_list:
        image = item[0]
        label = item[1]

        # Standardize the image
        standardized_im = standardize_input(image)

        # Create a numerical label
        binary_label = encode(label)

        # Append the image, and it's one hot encoded label to the
full, processed list of image data
        standard_list.append((standardized_im, binary_label))

    return standard_list

# Standardize all training images
STANDARDIZED_LIST = preprocess(IMAGE_LIST)

# Display a standardized image and its label

# Select an image by index

image_num = 1
selected_image = STANDARDIZED_LIST[image_num][0]
selected_label = STANDARDIZED_LIST[image_num][1]


# Display image and data about it
plt.imshow(selected_image)
print("Shape: "+str(selected_image.shape))
print("Label [1 = cloud, 0 = noncloud]: " + str(selected_label))
```

```
Shape: (600, 1100, 4)
Label [1 = cloud, 0 = noncloud]: 1
```



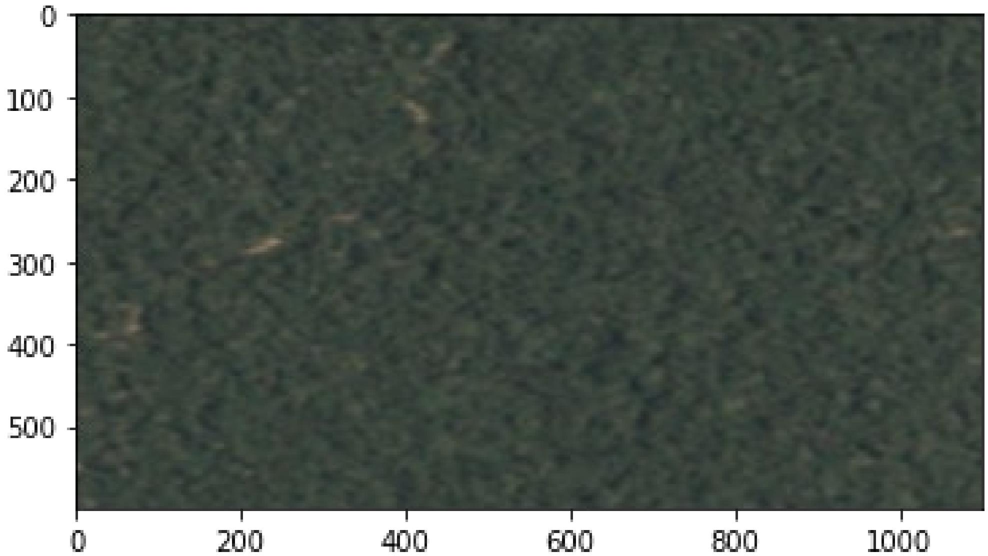
Step 3: Feature Extraction

Let's try to create a feature that represents the brightness in an image. We'll be extracting the **average brightness** using HSV colorspace. Specifically, we'll use the V channel (a measure of brightness), add up the pixel values in the V channel, then divide that sum by the area of the image to get the average Value of the image.

```
# Find the average Value or brightness of an image
def avg_brightness(rgb_image):
    # Convert image to HSV
    hsv = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2HSV)

    # Add up all the pixel values in the V channel
    sum_brightness = np.sum(hsv[2:,2:,2])
    area = 600*1100.0  # pixels

    # find the avg
    avg = sum_brightness/area

    return avg

# Testing average brightness levels
# Look at a number of different cloud and noncloud images and think
about
# what average brightness value separates the two types of images

# As an example, a "noncloud" image is loaded in and its avg
brightness is displayed
image_num = 190
test_im = STANDARDIZED_LIST[image_num][0]

avg = avg_brightness(test_im)
```

```
print('Avg brightness: ' + str(avg))
plt.imshow(test_im)
```

```
Avg brightness: 66.9299590909091

<matplotlib.image.AxesImage at 0x7fc93f4b7bd0>
```



Step 4: Build the classifier

We'll turn our average brightness feature into a classifier that takes in a standardized image and returns a `predicted_label` for that image. This `estimate_label` function should return a value: 0 or 1 (cloud or non cloud, respectively).

```
# This function should take in RGB image input
def estimate_label(rgb_image, threshold):

    # Extract average brightness feature from an RGB image
    avg = avg_brightness(rgb_image)

    # Use the avg brightness feature to predict a label (0, 1)
    predicted_label = 0
    #threshold = 120
    if(avg > threshold):
        # if the average brightness is above the threshold value, we
classify it as "cloud"
        predicted_label = 1
    # else, the pred-cted_label can stay 0 (it is predicted to be
"noncloud")

    return predicted_label
```

Step 5: Evaluate the Classifier and Optimize

Here is where we test your classification algorithm using our test set of data that we set aside at the beginning of the notebook! Below, we load in the test dataset, standardize it using the `standardize` function you defined above, and then **shuffle** it; this ensures that order will not play a role in testing accuracy.

```
# Using the load_dataset function in helpers.py
# Load test data
TEST_IMAGE_LIST = load_dataset(image_dir_test)

# Standardize the test data
STANDARDIZED_TEST_LIST = preprocess(TEST_IMAGE_LIST)

# Shuffle the standardized test data
random.shuffle(STANDARDIZED_TEST_LIST)

# Constructs a list of misclassified images given a list of test
images and their labels
def get_misclassified_images(test_images, threshold):
    # Track misclassified images by placing them into a list
    misclassified_images_labels = []

    # Iterate through all the test images
    # Classify each image and compare to the true label
    for image in test_images:

        # Get true data
        im = image[0]
        true_label = image[1]

        # Get predicted label from your classifier
        predicted_label = estimate_label(im, threshold)

        # Compare true and predicted labels
        if(predicted_label != true_label):
            # If these labels are not equal, the image has been
misclassified
            misclassified_images_labels.append((im, predicted_label,
true_label))

    # Return the list of misclassified [image, predicted_label,
true_label] values
    return misclassified_images_labels

# Find all misclassified images in a given test set
MISCLASSIFIED = get_misclassified_images(STANDARDIZED_TEST_LIST,
threshold=99)

# Accuracy calculations
```

```
total = len(STANDARDIZED_TEST_LIST)
num_correct = total - len(MISCLASSIFIED)
accuracy = num_correct/total

print('Accuracy: ' + str(accuracy))
print("Number of misclassified images = " + str(len(MISCLASSIFIED)) +'
out of '+ str(total))
```

```
Accuracy: 0.7
Number of misclassified images = 6 out of 20
```

Conclusion

We received an accuracy of **70%** by using only one feature extraction, i.e the average brightness of the image. We could work more on this, for example features that involve the other 2 Hue and Saturation channels to extract more features. This concludes that most simple problems can be solved using traditional image processing, and doesn't need Advanced Deep Learning concepts always.

Tasks:

Task 1: Execute the above code properly.

Task 2: Now, implement the image classification with given dataset using machine learning. Also, calculate the accuracy of the model.

Task 3: Understand and explain what did you analyze. Make a detailed analysis report. Also, cover all of the following questions.

1) explain when we should use Machine Learning and when not.

2) Compare the accuracy of both the way. Also, explain why you are getting the difference in accuracy.

3) Why we use Machine Learning rather than just software development?

4) How do you improve the performance of a model and why we need to improve the model performance?

(Submit the PDF of the report)

Task 4: Implement the above image classification code without the use of machine learning, but use different dataset.

Task 5: Implement any other code such as prediction without the use of machine learning.