CSCI446/946 Big Data Analytics - Week 9

Lab 8 – Image Processing with Python

Introduction

This instruction is for the use of the Python programming language. In Lab8, you will practice image processing which correlates to big data analytics lifecycle phase 2-5: Data Preparation, Model Planning, Model Building and Communicate Results. To complete this lab, you are required to

1. download `Lab8.py` and the data from Moodle
2. Create a file `Lab8.py` to implement the code shown in `blue`
3. Create a Word document to write down your report by 1) explaining what you will do next, and 2) explain findings
4. format your report: title, heading, body of text, code, programming results, tables, figures, references, etc.

Task1: Image Data Analysis Using Python

Original Post on page and page

This tutorial takes a look at how to import images and observe it's properties, split the layers, and also looks at greyscale. The contents include an introduction about Pixel, observe basic properties of image, greyscale, use logical operator to process Pixel values, masking, and image processing.

```
# import packages
import imageio
import matplotlib.pyplot as plt
import numpy as np
import random

%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
```

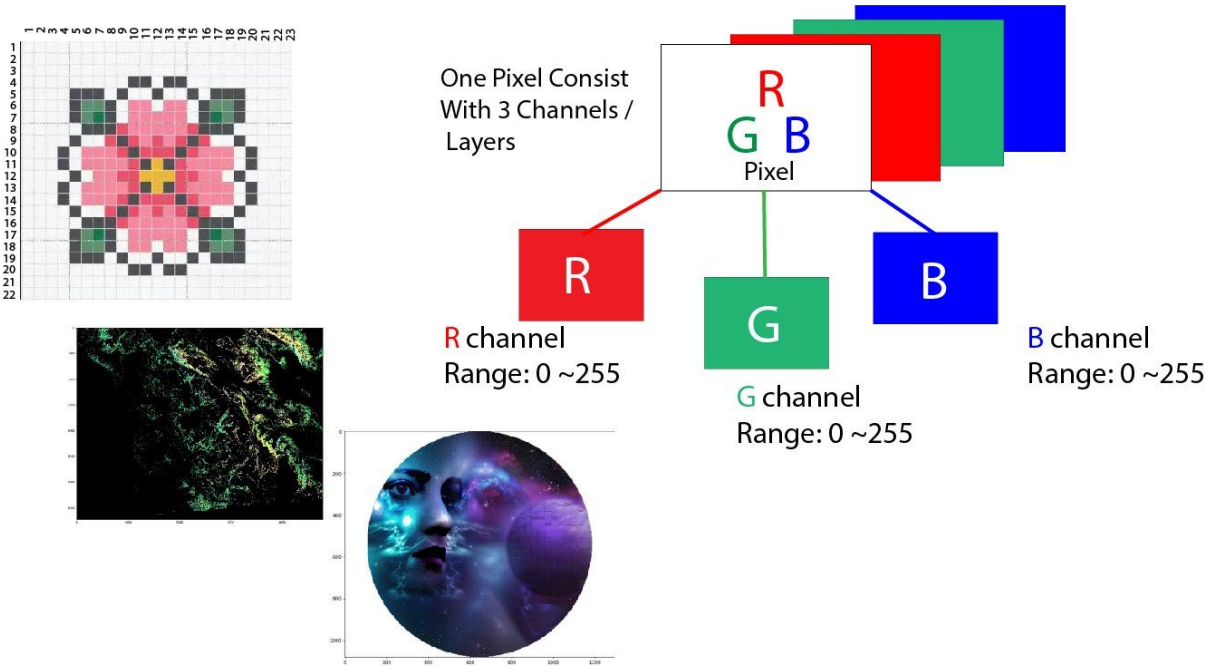
Introduction: A Little Bit About Pixel

Computers store images as a mosaic of tiny squares called pixels. Now, if these squares are too big or too few then we cannot represent smooth edges and curves or represent details. The more and smaller tiles we use, the smoother (less pixelated) the image will be. This refers to the resolution of the images.

The word pixel means a picture element. A single mosaic of pixels can represent images in black-and-white or in gray depending on whether pixel values are binary or integer numbers. Having multiple mosaics enables images to be of multiple colors, A common way to describe color pixel is by using three mosaics of pixels to represent the primary colors, namely Red,

Green, Blue. The pixel values are then the combination of the three values in the three mosaics. Images formed in this way are called RGB images.

Every photograph, in digital form, is made up of pixels. They are the smallest unit of information that makes up a picture. Usually round or square, they are typically arranged in a 2-dimensional grid.



Pixel values are often stored as byte values. Thus, if all three values are at full intensity, that means they're 255. It then shows as white, and if all three colors are muted, or has the value of 0, the pixel is black. The combination of these three will, in turn, give us a specific shade of the pixel color. Since each number is an 8-bit number, the values range from 0–255.

The combination of these three colors tends to the highest value among them. Since each value can have 256 different intensity or brightness value, it makes 16.8 million total shades.

Now let's load an image and observe its various properties in general.

```
pic = imageio.imread("data/logic_on_pic.jpg")
plt.figure(figsize = (5,5))
```

```
plt.imshow(pic)
```

Observe Basic Properties of Image

```
print('Type of the image : ' , type(pic))
print('Shape of the image : {}'.format(pic.shape))
print('Image Hight {}'.format(pic.shape[0]))
print('Image Width {}'.format(pic.shape[1]))
print('Dimension of Image {}'.format(pic.ndim))
```

The shape of the ndarray shows that it is a three-layered matrix. The first two numbers here are length and width, and the third number (i.e. 3) is for three layers: Red, Green, Blue. So, if we calculate the size of an RGB image, the total size will be counted as height x width x 3

```
print('Image size {}'.format(pic.size))
print('Maximum RGB value in this image {}'.format(pic.max()))
print('Minimum RGB value in this image {}'.format(pic.min()))

# A specific pixel located at Row : 100 ; Column : 50
# Each channel's value of it, gradually R , G , B
print('Value of only R channel {}'.format(pic[ 100, 50, 0]))
print('Value of only G channel {}'.format(pic[ 100, 50, 1]))
print('Value of only B channel {}'.format(pic[ 100, 50, 2]))
```

Now let's take a quick view of each channel in the whole image.

```
plt.title('R channel')
plt.ylabel('Height {}'.format(pic.shape[0]))
plt.xlabel('Width {}'.format(pic.shape[1]))
plt.imshow(pic[ : , : , 0])
plt.show()

plt.title('G channel')
plt.ylabel('Height {}'.format(pic.shape[0]))
plt.xlabel('Width {}'.format(pic.shape[1]))
plt.imshow(pic[ : , : , 1])
plt.show()

plt.title('B channel')
plt.ylabel('Height {}'.format(pic.shape[0]))
plt.xlabel('Width {}'.format(pic.shape[1]))
plt.imshow(pic[ : , : , 2])
plt.show()
```

Now, we can also able to change the number of RGB values. As an example, let's set the Red, Green, Blue layer for following Rows values to full intensity.

- R channel: Row — 50 to 60
- G channel: Row — 100 to 110
- B channel: Row — 150 to 160

We'll load the image once so that we can visualize each change simultaneously.

```
# R channel: Row – 50 to 60
# full intensity to those pixel's R channel
pic[50:60 , : , 0] = 255
plt.figure( figsize = (5,5))
plt.imshow(pic)
plt.show()

# G channel: Row – 100 to 110
# full intensity to those pixel's G channel
pic[100:110 , : , 1] = 255
plt.figure( figsize = (5,5))
plt.imshow(pic)
plt.show()

# B channel: Row – 150 to 160
# full intensity to those pixel's B channel
pic[150:160 , : , 2] = 255
plt.figure( figsize = (5,5))
plt.imshow(pic)
plt.show()
```

To make it more clear let's change the column section too and this time we'll change the RGB channel simultaneously.

```
# set value 200 of all channels to those pixels which turns
them to white
pic[ 50:160 , 200:250 , [0,1,2] ] = 200
plt.figure( figsize = (5,5))
plt.imshow(pic)
plt.show()
```

Now, we know that each pixel of the image is represented by three integers. Splitting the image into separate color components is just a matter of pulling out the correct slice of the image array.

```
pic = imageio.imread('data/logic_on_pic.jpg')
fig, ax = plt.subplots(nrows = 1, ncols=3, figsize=(15,5))
for c, ax in zip(range(3), ax):
    ax.imshow(pic[ :, :, c])
```

We can also create histograms for full colour images. A program to create colour histograms starts in a familiar way:

```
# tuple to select colors of each channel line
colors = ("red", "green", "blue")
channel_ids = (0, 1, 2)

# create the histogram plot, with three lines, one for
# each color
plt.figure()
```

```
plt.xlim([0, 256])
for channel_id, c in zip(channel_ids, colors):
    histogram, bin_edges = np.histogram(
        pic[:, :, channel_id], bins=128, range=(0, 256)
    )
    plt.plot(bin_edges[0:-1], histogram, color=c)

plt.title("Color Histogram")
plt.xlabel("Color value")
plt.ylabel("Pixel count")
```

Greyscale

Black and white images are stored in 2-Dimensional arrays. There're two types of black and white images:

- Binary: Pixel is either black or white:0 or 255
- Greyscale: Ranges of shades of grey:0 ~ 255

Greyscaling is a process by which an image is converted from a full color to shades of grey. In image processing tools, for example: in OpenCV, many functions use greyscale images before processing, and this is done because it simplifies the image, acting almost as noise reduction and increasing processing time as there's less information in the images.

There are a couple of ways to do this in python to convert an image to grayscale, but a straightforward way of using matplotlib is to take the weighted mean of the RGB value of original image using this formula.

Y' = 0.299 R + 0.587 G + 0.114 B

```
pic = imageio.imread('data/logic_on_pic.jpg')
gray = lambda rgb : np.dot(rgb[... , :3] , [0.299 , 0.587,
0.114])
gray = gray(pic)
plt.figure( figsize = (5,5))
plt.imshow(gray, cmap = plt.get_cmap(name = 'gray'))
plt.show()
```

Skimage does not provide a special function to compute histograms, but we can use the function np.histogram instead:

```
# create the histogram
histogram, bin_edges = np.histogram(gray, bins=5, range=(0,
1))
# configure and draw the histogram figure
plt.figure()
plt.title("Grayscale Histogram")
plt.xlabel("grayscale value")
plt.ylabel("pixel count")
plt.plot(bin_edges[0:-1], histogram)  # <- or here
```

Use Logical Operator To Process Pixel Values

We can create a ndarray in the same size by using a logical operator. However, this won't create any new arrays, but it simply returns True to its host variable. For example, let's consider we want to filter out some low-value pixels or high-value or (any condition) in an RGB image, and yes, it would be great to convert RGB to grayscale, but for now, we won't go for that rather than deal with a color image.

Let's first load an image and show it on screen.

```
pic = imageio.imread('data/logic_on_pic.jpg')
plt.figure(figsize=(5,5))
plt.imshow(pic)
plt.show()
```

let's consider this dump image. Now, for any case, we want to filter out all the pixel values, which is below than, let's assume, 20. For this, we'll use a logical operator to do this task, which we'll return as a value of True for all the index.

```
low_pixel = pic < 20
# to ensure of it let's check if all values in low_pixel are
True or not
if low_pixel.any() == True:
    print(low_pixel.shape)
```

Now as we said, a host variable is not traditionally used, but I refer it because it behaves. It just holds the True value and nothing else. So, if we see the shape of both low_pixel and pic , we'll find that both have the same shape.

```
print(pic.shape)
print(low_pixel.shape)
```

We generated that low-value filter using a global comparison operator for all the values less than 200. However, we can use this low_pixel array as an index to set those low values to some specific values, which may be higher than or lower than the previous pixel value.

```
pic = imageio.imread('data/logic_on_pic.jpg')

# set value randomly range from 25 to 225 - these value also
randomly choosen
pic[low_pixel] = random.randint(25,225)
# display the image
plt.figure( figsize = (5,5))
plt.imshow(pic)
plt.show()
```

Masking

Image masking is an image processing technique that is used to remove the background from which photographs those have fuzzy edges, transparent or hair portions.

Now, we'll create a mask that is in shape of a circular disc. First, we'll measure the distance from the center of the image to every border pixel values. And we take a convenient radius value, and then using logical operator, we'll create a circular disc. It's quite simple, let's see the code.

```
# Load the image
pic = imageio.imread('data/logic_on_pic.jpg')
# seperate the row and column values
total_row , total_col , layers = pic.shape
'''     Create vector.     Ogrid is a compact method of
creating a multidimensional
ndarray operations in single lines.
for ex:
>>> np.ogrid[0:5,0:5]
output: [array([[0],
                [1],
                [2],
                [3],
                [4]]),
         array([[0, 1, 2, 3, 4]])]
'''
x , y = np.ogrid[:total_row , :total_col]
# get the center values of the image
cen_x , cen_y = total_row/2 , total_col/2
'''
 Measure distance value from center to each border pixel.
 To make it easy, we can think it's like, we draw a line from
center-
 to each edge pixel value --> s**2 = (Y-y)**2 + (X-x)**2
'''
distance_from_the_center = np.sqrt((x-cen_x)**2 + (y-
cen_y)**2)
# Select convenient radius value
radius = (total_row/2)
# Using logical operator '>'
'''
logical operator to do this task which will return as a value
of True
for all the index according to the given condition
'''
circular_pic = distance_from_the_center > radius
'''
let assign value zero for all pixel value that outside the
cirular
disc. All the pixel value outside the circular disc, will be
black
now.
'''
```

```
pic[circular_pic] = 0
plt.figure(figsize = (5,5))
plt.imshow(pic)
plt.show()
```

Image Processing

There's something interesting about this image. Like many other visualizations, the colors in each RGB layer mean something. For example, the intensity of the red will be an indication of altitude of the geographical data point in the pixel. The intensity of blue will indicate a measure of aspect, and the green will indicate slope. These colors will help communicate this information in a quicker and more effective way rather than showing numbers.

Red pixel indicates: Altitude
Blue pixel indicates: Aspect
Green pixel indicates: Slope

There is, by just looking at this colorful image, a trained eye that can tell already what the altitude is, what the slope is, and what the aspect is. So, that's the idea of loading some more meaning to these colors to indicate something more scientific.

```
# Only Red Pixel value , higher than 180
pic = imageio.imread('data/logic_on_pic.jpg')
red_mask = pic[:, :, 0] < 180
pic[red_mask] = 0
plt.figure(figsize=(5,5))
plt.imshow(pic)

# Only Green Pixel value , higher than 180
pic = imageio.imread('data/logic_on_pic.jpg')
green_mask = pic[:, :, 1] < 180
pic[green_mask] = 0
plt.figure(figsize=(5,5))
plt.imshow(pic)
# Only Blue Pixel value , higher than 180
pic = imageio.imread('data/logic_on_pic.jpg')
blue_mask = pic[:, :, 2] < 180
pic[blue_mask] = 0
plt.figure(figsize=(5,5))
plt.imshow(pic)
# Composite mask using logical_and
pic = imageio.imread('data/logic_on_pic.jpg')
final_mask = np.logical_and(red_mask, green_mask, blue_mask)
pic[final_mask] = 40
plt.figure(figsize=(5,5))
plt.imshow(pic)
```

Task2: CNN for Image Representation

Original Post on page

```
# import packages
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Conv2D, MaxPool2D,
Flatten
try:
    from keras.utils.np_utils import to_categorical
except:
    from keras.utils import to_categorical

from sklearn.metrics import accuracy_score

import warnings
warnings.filterwarnings("ignore")
```

Convolutional neural networks (CNN) – the concept behind recent breakthroughs and developments in deep learning.

CNNs have broken the mold and ascended the throne to become the state-of-the-art computer vision technique. Among the different types of neural networks (others include recurrent neural networks (RNN), long short term memory (LSTM), artificial neural networks (ANN), etc.), CNNs are easily the most popular.

These convolutional neural network models are ubiquitous in the image data space. They work phenomenally well on computer vision tasks like image classification, object detection, image recognition, etc.

There are various datasets that you can leverage for applying convolutional neural networks. Here we use the most popular dataset: MNIST (Hand-written Digits).

MNIST comes with Keras by default and you can simply load the train and test files using a few lines of code:

```
# loading dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# let's print the shape of the dataset
print("X_train shape", X_train.shape)
```

```
print("y_train shape", y_train.shape)
print("X_test shape", X_test.shape)
print("y_test shape", X_test.shape)
```

Before we train a CNN model, let's build a basic Fully Connected Neural Network for the dataset. The basic steps to build an image classification model using a neural network are:

1. Flatten the input image dimensions to 1D (width pixels x height pixels)
2. Normalize the image pixel values (divide by 255)
3. One-Hot Encode the categorical column
4. Build a model architecture (Sequential) with Dense layers
5. Train the model and make predictions

Here's how you can build a neural network model for MNIST. I have commented on the relevant parts of the code for better understanding:

```
# Flattening the images from the 28x28 pixels to 1D 787 pixels
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# normalizing the data to help with the training
X_train /= 255
X_test /= 255

# one-hot encoding using keras' numpy-related utilities
n_classes = 10
print("Shape before one-hot encoding: ", y_train.shape)
Y_train = to_categorical(y_train, n_classes)
Y_test = to_categorical(y_test, n_classes)
print("Shape after one-hot encoding: ", Y_train.shape)

# building a linear stack of layers with the sequential model
model = Sequential()
# hidden layer
model.add(Dense(100, input_shape=(784,), activation='relu'))
# output layer
model.add(Dense(10, activation='softmax'))

# looking at the model summary
model.summary()
# compiling the sequential model
model.compile(loss='categorical_crossentropy',
metrics=['accuracy'], optimizer='adam')
# training the model for 10 epochs
model.fit(X_train, Y_train, batch_size=128, epochs=10,
validation_data=(X_test, Y_test))
```

After running the above code, you'd realized that we are getting a good validation accuracy of around 98%.

Let's modify the above code to build a CNN model.

One major advantage of using CNNs over NNs is that you do not need to flatten the input images to 1D as they are capable of working with image data in 2D. This helps in retaining the "spatial" properties of images.

The complete code for the CNN model is as follows:

```
# loading the dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# building the input vector from the 28x28 pixels
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# normalizing the data to help with the training
X_train /= 255
X_test /= 255

# one-hot encoding using keras' numpy-related utilities
n_classes = 10
print("Shape before one-hot encoding: ", y_train.shape)
Y_train = to_categorical(y_train, n_classes)
Y_test = to_categorical(y_test, n_classes)
print("Shape after one-hot encoding: ", Y_train.shape)

# building a linear stack of layers with the sequential model
model = Sequential()
# convolutional layer
model.add(Conv2D(25, kernel_size=(3,3), strides=(1,1),
padding='valid', activation='relu', input_shape=(28,28,1)))
model.add(MaxPool2D(pool_size=(1,1)))
# flatten output of conv
model.add(Flatten())
# hidden layer
model.add(Dense(100, activation='relu'))
# output layer
model.add(Dense(10, activation='softmax'))

# compiling the sequential model
model.compile(loss='categorical_crossentropy',
metrics=['accuracy'], optimizer='adam')

# training the model for 10 epochs
model.fit(X_train, Y_train, batch_size=128, epochs=10,
validation_data=(X_test, Y_test))
```