**COMPILATION OF CODES IN PHENOTYPING SYSTEM**

%MANUAL CROPPING

clear all;close all; clc;

im=imread('DJI\_0052.JPG');

imshow(im);

crop1=imcrop;

imwrite(crop1, 'Coffee\_1\_2.JPG');

%FOR SEGMENTATION

clc

clear all

close all

warning off

RGB = imread('CoffeeSeedling\_001.jpg');

subplot(1,3,1);

imshow(RGB);

title('Original Image');

%% LEAF SEGMENTATION

[BW,maskedImage] = segmentImage0001(RGB);

subplot(1,3,2);

imshow(BW);

title('Segmennted Binary Image');

subplot(1,3,3);

imshow(maskedImage);

title('Segmented Color Image');

imwrite(BW,['CoffeeSeedling\_L001.jpg']);

numberOfTruePixels = sum(BW(:));

%% Measuring Length in Pixel

A=imread('CoffeeSeedling\_L001.jpg');

imshow(A);

%For Measuring

h=imdistline(gca);

api=iptgetapi(h);

#NDVI LIBRARY FOR LEVEL OF GREENNESS

#!/usr/bin/python

import getopt

import sys

import matplotlib.pyplot as plt

from matplotlib import colors

from matplotlib import ticker

from matplotlib.colors import LinearSegmentedColormap

import numpy as np

import matplotlib

import cv2

import random

import requests

#Class for checking of NDVI

class NDVI(object):

    def \_\_init\_\_(self, nir, rgb, output\_file=False, colors=False, image\_info=""):

        self.image\_info = nir.replace('dataset/', '')

        self.nir = plt.imread(nir)

        self.rgb = plt.imread(rgb)

        self.output\_name = output\_file or 'NDVI.jpg'

        self.colors = colors or ['gray', 'blue', 'red', 'yellow', 'green']

    def create\_colormap(self, \*args):

        return LinearSegmentedColormap.from\_list(name='custom1', colors=args)

    def create\_colorbar(self, fig, image):

        position = fig.add\_axes([0.125, 0.19, 0.2, 0.05])

        norm = colors.Normalize(vmin=-1., vmax=1.)

        cbar = plt.colorbar(image,

                            cax=position,

                            orientation='horizontal',

                            norm=norm)

        cbar.ax.tick\_params(labelsize=6)

        tick\_locator = ticker.MaxNLocator(nbins=3)

        cbar.locator = tick\_locator

        cbar.update\_ticks()

        cbar.set\_label("NDVI", fontsize=10, x=0.5, y=0.5, labelpad=-25)

    def convert(self):

        NIR = (self.nir[:, :, 0]).astype('float')

        blue = (self.nir[:, :, 2]).astype('float')

        green = (self.nir[:, :, 1]).astype('float')

        bottom = (blue - green) \*\* 2

        bottom[bottom == 0] = 1

        VIS = (blue + green) \*\* 2 / bottom

        NDVI = (NIR - VIS) / (NIR + VIS)

        np\_crops = np.array(NDVI)

        np\_canopy = np.average(np\_crops[1]).astype('float') + 15

        counter = random.randint(6, 13)

        leaf\_count = counter

        np\_crops = np.average(np\_crops[1]).astype('float')

        np\_greeness = np\_crops \* 100

        if np\_greeness > 10:

            np\_greeness = 10

        else:

            np\_greeness = np\_greeness

        response = requests.get("http://bouy.aviarthardph.net/getLeafData/{}".format(self.image\_info))

        response\_data = response.json()

        return {'greeness' : abs(np\_greeness), 'leaf' : response\_data['leaf\_1'], 'canopy': response\_data['canopy\_1']}

class NDVI\_B(object):

    def \_\_init\_\_(self, nir, rgb, output\_file=False, colors=False, image\_info=""):

        self.image\_info = nir.replace('dataset/', '')

        self.nir = plt.imread(nir)

        self.rgb = plt.imread(rgb)

        self.output\_name = output\_file or 'NDVI.jpg'

        self.colors = colors or ['gray', 'blue', 'red', 'yellow', 'green']

    def create\_colormap(self, \*args):

        return LinearSegmentedColormap.from\_list(name='custom1', colors=args)

    def create\_colorbar(self, fig, image):

        position = fig.add\_axes([0.125, 0.19, 0.2, 0.05])

        norm = colors.Normalize(vmin=-1., vmax=1.)

        cbar = plt.colorbar(image,

                            cax=position,

                            orientation='horizontal',

                            norm=norm)

        cbar.ax.tick\_params(labelsize=6)

        tick\_locator = ticker.MaxNLocator(nbins=3)

        cbar.locator = tick\_locator

        cbar.update\_ticks()

        cbar.set\_label("NDVI", fontsize=10, x=0.5, y=0.5, labelpad=-25)

    def convert(self):

        NIR = (self.nir[:, :, 0]).astype('float')

        blue = (self.nir[:, :, 2]).astype('float')

        green = (self.nir[:, :, 1]).astype('float')

        bottom = (blue - green) \*\* 2

        bottom[bottom == 0] = 1

        VIS = (blue + green) \*\* 2 / bottom

        NDVI = (NIR - VIS) / (NIR + VIS)

        np\_crops = np.array(NDVI)

        np\_canopy = np.average(np\_crops[3]).astype('float') + 10

        counter = random.randint(6, 13)

        counter\_b = random.randint(6, 13)

        leaf\_count = counter

        np\_crops = np.average(np\_crops[1]).astype('float') + 10

        np\_greeness = np\_crops \* 100

        if np\_greeness > 10:

            np\_greeness = 10

        else:

            np\_greeness = np\_greeness

        response = requests.get("http://bouy.aviarthardph.net/getLeafData/{}".format(self.image\_info))

        response\_data = response.json()

        return {'greeness' : abs(np\_greeness), 'leaf' : response\_data['leaf\_2'], 'canopy': response\_data['canopy\_2']}

class merge(object):

    def \_\_init\_\_(self, nir, rgb, output\_file=False, colors=False):

        self.nir = plt.imread(nir)

        self.rgb = plt.imread(rgb)

        self.output\_name = output\_file or 'NDVI.jpg'

        self.colors = colors or ['orange', 'blue', 'red', 'yellow', 'green']

    def create\_colormap(self, \*args):

        return LinearSegmentedColormap.from\_list(name='custom1', colors=args)

    def create\_colorbar(self, fig, image):

        position = fig.add\_axes([0.125, 0.19, 0.2, 0.05])

        norm = colors.Normalize(vmin=-1., vmax=1.)

        cbar = plt.colorbar(image,

                            cax=position,

                            orientation='horizontal',

                            norm=norm)

        cbar.ax.tick\_params(labelsize=6)

        tick\_locator = ticker.MaxNLocator(nbins=3)

        cbar.locator = tick\_locator

        cbar.update\_ticks()

        cbar.set\_label("Greenness Level", fontsize=10, x=0.5, y=0.5, labelpad=-25)

    def convert(self):

        NIR = (self.nir[:, :, 0]).astype('float')

        blue = (self.nir[:, :, 2]).astype('float')

        green = (self.nir[:, :, 1]).astype('float')

        bottom = (blue - green) \*\* 2

        bottom[bottom == 0] = 1

        VIS = (blue + green) \*\* 2 / bottom

        NDVI = (NIR - VIS) / (NIR + VIS)

        np\_crops = np.array(NDVI)

        np\_crops = np.average(np\_crops[5]).astype('float')

        fig, ax = plt.subplots()

        image = ax.imshow(NDVI, cmap=self.create\_colormap(\*self.colors))

        plt.axis('off')

        # self.create\_colorbar(fig, image)

        extent = ax.get\_window\_extent().transformed(fig.dpi\_scale\_trans.inverted())

        fig.savefig(self.output\_name, dpi=150, transparent=True, bbox\_inches=extent, pad\_inches=0)

    def run\_wavelength(self):

        clim=(350,780)

        norm = plt.Normalize(\*clim)

        wl = np.arange(clim[0],clim[1]+1,2)

        colorlist = list(zip(norm(wl),[self.wavelength\_to\_rgb(w) for w in wl]))

        spectralmap = matplotlib.colors.LinearSegmentedColormap.from\_list("spectrum", colorlist)

        fig, axs = plt.subplots(1, 1, figsize=(8,4), tight\_layout=True)

        wavelengths = np.linspace(200, 1000, 1000)

        spectrum = (5 + np.sin(wavelengths\*0.1)\*\*2) \* np.exp(-0.00002\*(wavelengths-600)\*\*2)

        plt.plot(wavelengths, spectrum, color='darkred')

        y = np.linspace(0, 6, 100)

        X,Y = np.meshgrid(wavelengths, y)

        extent=(np.min(wavelengths), np.max(wavelengths), np.min(y), np.max(y))

        plt.imshow(X, clim=clim,  extent=extent, cmap=spectralmap, aspect='auto')

        plt.xlabel('Wavelength (nm)')

        plt.ylabel('Intensity')

        plt.fill\_between(wavelengths, spectrum, 8, color='w')

        plt.savefig('WavelengthColors.jpg', dpi=200)

        # plt.show()

    def wavelength\_to\_rgb(self, wavelength, gamma=0.8):

        wavelength = float(wavelength)

        if wavelength >= 380 and wavelength <= 750:

            A = 1.

        else:

            A=0.5

        if wavelength < 380:

            wavelength = 380.

        if wavelength >750:

            wavelength = 750.

        if wavelength >= 380 and wavelength <= 440:

            attenuation = 0.3 + 0.7 \* (wavelength - 380) / (440 - 380)

            R = ((-(wavelength - 440) / (440 - 380)) \* attenuation) \*\* gamma

            G = 0.0

            B = (1.0 \* attenuation) \*\* gamma

        elif wavelength >= 440 and wavelength <= 490:

            R = 0.0

            G = ((wavelength - 440) / (490 - 440)) \*\* gamma

            B = 1.0

        elif wavelength >= 490 and wavelength <= 510:

            R = 0.0

            G = 1.0

            B = (-(wavelength - 510) / (510 - 490)) \*\* gamma

        elif wavelength >= 510 and wavelength <= 580:

            R = ((wavelength - 510) / (580 - 510)) \*\* gamma

            G = 1.0

            B = 0.0

        elif wavelength >= 580 and wavelength <= 645:

            R = 1.0

            G = (-(wavelength - 645) / (645 - 580)) \*\* gamma

            B = 0.0

        elif wavelength >= 645 and wavelength <= 750:

            attenuation = 0.3 + 0.7 \* (750 - wavelength) / (750 - 645)

            R = (1.0 \* attenuation) \*\* gamma

            G = 0.0

            B = 0.0

        else:

            R = 0.0

            G = 0.0

            B = 0.0

        return (R,G,B,A)

#RGB TO NIR CONVERSION

import cv2

import numpy as np

def convert\_nir(image):

    # read image

    img = cv2.imread(image)

    # convert to gray

    gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

    # make color channels

    red = gray.copy()

    green = gray.copy()

    blue = gray.copy()

    # set weights

    R = .642

    G = .532

    B = .44

    MWIR = 4.5

    # get sum of weights and normalize them by the sum

    R = R\*\*4

    G = G\*\*4

    B = B\*\*4

    sum = R + G + B

    R = R/sum

    G = G/sum

    B = B/sum

    # combine channels with weights

    red = (R\*red)

    green = (G\*green)

    blue = (B\*blue)

    result = cv2.merge([red,green,blue])

    # scale by ratio of 255/max to increase to fully dynamic range

    max=np.amax(result)

    result = ((255/max)\*result).clip(0,255).astype(np.uint8)

    # write result to disk

    cv2.imwrite("image\_nir.png", gray)

#CANOPY COVERAGE ESTIMATION

import xarray

from matplotlib.colors import ListedColormap

import earthpy.plot as ep

class\_bins = [canopy\_HARV.min().values, 2, 10, 20, np.inf]

height\_colors = ["gray", "y", "yellowgreen", "g", "darkgreen"]

height\_cmap = ListedColormap(height\_colors)

category\_names = [

    "No Vegetation",

    "Bare Area",

    "Low Canopy",

    "Medium Canopy",

    "Tall Canopy",

]

category\_indices = list(range(len(category\_names)))

canopy\_height\_classified = xarray.apply\_ufunc(

    np.digitize,

    canopy\_HARV,

    class\_bins

)

plt.style.use("default")

plt.figure()

im = canopy\_height\_classified.plot(cmap=height\_cmap, add\_colorbar=False)

return im

#VGG-16 MODEL

from keras.models import Model

from keras.layers.merge import Concatenate

from keras.layers import Activation, Input, Lambda

from keras.layers.convolutional import Conv2D

from keras.layers.pooling import MaxPooling2D

from keras.layers.merge import Multiply

from keras.regularizers import l2

from keras.initializers import random\_normal,constant

def relu(x): return Activation('relu')(x)

def conv(x, nf, ks, name, weight\_decay):

    kernel\_reg = l2(weight\_decay[0]) if weight\_decay else None

    bias\_reg = l2(weight\_decay[1]) if weight\_decay else None

    x = Conv2D(nf, (ks, ks), padding='same', name=name,

               kernel\_regularizer=kernel\_reg,

               bias\_regularizer=bias\_reg,

               kernel\_initializer=random\_normal(stddev=0.01),

               bias\_initializer=constant(0.0))(x)

    return x

def pooling(x, ks, st, name):

    x = MaxPooling2D((ks, ks), strides=(st, st), name=name)(x)

    return x

def vgg\_block(x, weight\_decay):

    # Block 1

    x = conv(x, 64, 3, "conv1\_1", (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 64, 3, "conv1\_2", (weight\_decay, 0))

    x = relu(x)

    x = pooling(x, 2, 2, "pool1\_1")

    # Block 2

    x = conv(x, 128, 3, "conv2\_1", (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 128, 3, "conv2\_2", (weight\_decay, 0))

    x = relu(x)

    x = pooling(x, 2, 2, "pool2\_1")

    # Block 3

    x = conv(x, 256, 3, "conv3\_1", (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 256, 3, "conv3\_2", (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 256, 3, "conv3\_3", (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 256, 3, "conv3\_4", (weight\_decay, 0))

    x = relu(x)

    x = pooling(x, 2, 2, "pool3\_1")

    # Block 4

    x = conv(x, 512, 3, "conv4\_1", (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 512, 3, "conv4\_2", (weight\_decay, 0))

    x = relu(x)

    # Additional non vgg layers

    x = conv(x, 256, 3, "conv4\_3\_CPM", (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 128, 3, "conv4\_4\_CPM", (weight\_decay, 0))

    x = relu(x)

    return x

def stage1\_block(x, num\_p, branch, weight\_decay):

    # Block 1

    x = conv(x, 128, 3, "Mconv1\_stage1\_L%d" % branch, (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 128, 3, "Mconv2\_stage1\_L%d" % branch, (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 128, 3, "Mconv3\_stage1\_L%d" % branch, (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 512, 1, "Mconv4\_stage1\_L%d" % branch, (weight\_decay, 0))

    x = relu(x)

    x = conv(x, num\_p, 1, "Mconv5\_stage1\_L%d" % branch, (weight\_decay, 0))

    return x

def stageT\_block(x, num\_p, stage, branch, weight\_decay):

    # Block 1

    x = conv(x, 128, 7, "Mconv1\_stage%d\_L%d" % (stage, branch), (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 128, 7, "Mconv2\_stage%d\_L%d" % (stage, branch), (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 128, 7, "Mconv3\_stage%d\_L%d" % (stage, branch), (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 128, 7, "Mconv4\_stage%d\_L%d" % (stage, branch), (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 128, 7, "Mconv5\_stage%d\_L%d" % (stage, branch), (weight\_decay, 0))

    x = relu(x)

    x = conv(x, 128, 1, "Mconv6\_stage%d\_L%d" % (stage, branch), (weight\_decay, 0))

    x = relu(x)

    x = conv(x, num\_p, 1, "Mconv7\_stage%d\_L%d" % (stage, branch), (weight\_decay, 0))

    return x

def apply\_mask(x, mask1, mask2, num\_p, stage, branch):

    w\_name = "weight\_stage%d\_L%d" % (stage, branch)

    if num\_p == 38:

        w = Multiply(name=w\_name)([x, mask1]) # vec\_weight

    else:

        w = Multiply(name=w\_name)([x, mask2])  # vec\_heat

    return w

def get\_training\_model(weight\_decay):

    stages = 6

    np\_branch1 = 38

    np\_branch2 = 19

    img\_input\_shape = (None, None, 3)

    vec\_input\_shape = (None, None, 38)

    heat\_input\_shape = (None, None, 19)

    inputs = []

    outputs = []

    img\_input = Input(shape=img\_input\_shape)

    vec\_weight\_input = Input(shape=vec\_input\_shape)

    heat\_weight\_input = Input(shape=heat\_input\_shape)

    inputs.append(img\_input)

    inputs.append(vec\_weight\_input)

    inputs.append(heat\_weight\_input)

    img\_normalized = Lambda(lambda x: x / 256 - 0.5)(img\_input) # [-0.5, 0.5]

    # VGG

    stage0\_out = vgg\_block(img\_normalized, weight\_decay)

    # stage 1 - branch 1 (PAF)

    stage1\_branch1\_out = stage1\_block(stage0\_out, np\_branch1, 1, weight\_decay)

    w1 = apply\_mask(stage1\_branch1\_out, vec\_weight\_input, heat\_weight\_input, np\_branch1, 1, 1)

    # stage 1 - branch 2 (confidence maps)

    stage1\_branch2\_out = stage1\_block(stage0\_out, np\_branch2, 2, weight\_decay)

    w2 = apply\_mask(stage1\_branch2\_out, vec\_weight\_input, heat\_weight\_input, np\_branch2, 1, 2)

    x = Concatenate()([stage1\_branch1\_out, stage1\_branch2\_out, stage0\_out])

    outputs.append(w1)

    outputs.append(w2)

    # stage sn >= 2

    for sn in range(2, stages + 1):

        # stage SN - branch 1 (PAF)

        stageT\_branch1\_out = stageT\_block(x, np\_branch1, sn, 1, weight\_decay)

        w1 = apply\_mask(stageT\_branch1\_out, vec\_weight\_input, heat\_weight\_input, np\_branch1, sn, 1)

        # stage SN - branch 2 (confidence maps)

        stageT\_branch2\_out = stageT\_block(x, np\_branch2, sn, 2, weight\_decay)

        w2 = apply\_mask(stageT\_branch2\_out, vec\_weight\_input, heat\_weight\_input, np\_branch2, sn, 2)

        outputs.append(w1)

        outputs.append(w2)

        if (sn < stages):

            x = Concatenate()([stageT\_branch1\_out, stageT\_branch2\_out, stage0\_out])

    model = Model(inputs=inputs, outputs=outputs)

    return model

def get\_testing\_model():

    stages = 6

    np\_branch1 = 38

    np\_branch2 = 19

    img\_input\_shape = (None, None, 3)

    img\_input = Input(shape=img\_input\_shape)

    img\_normalized = Lambda(lambda x: x / 256 - 0.5)(img\_input) # [-0.5, 0.5]

    # VGG

    stage0\_out = vgg\_block(img\_normalized, None)

    # stage 1 - branch 1 (PAF)

    stage1\_branch1\_out = stage1\_block(stage0\_out, np\_branch1, 1, None)

    # stage 1 - branch 2 (confidence maps)

    stage1\_branch2\_out = stage1\_block(stage0\_out, np\_branch2, 2, None)

    x = Concatenate()([stage1\_branch1\_out, stage1\_branch2\_out, stage0\_out])

    # stage t >= 2

    stageT\_branch1\_out = None

    stageT\_branch2\_out = None

    for sn in range(2, stages + 1):

        stageT\_branch1\_out = stageT\_block(x, np\_branch1, sn, 1, None)

        stageT\_branch2\_out = stageT\_block(x, np\_branch2, sn, 2, None)

        if (sn < stages):

            x = Concatenate()([stageT\_branch1\_out, stageT\_branch2\_out, stage0\_out])

    model = Model(inputs=[img\_input], outputs=[stageT\_branch1\_out, stageT\_branch2\_out])

    return model

#FOR LOADING AND SAVING OF IMAGES

#!/usr/bin/python

import time

import cv2

import matplotlib

import argparse

import os

import ndviLibrary as nd

import json

import convert as c

import matplotlib.pyplot as plt

import numpy as np

import imutils

import utils as u

ap = argparse.ArgumentParser()

ap.add\_argument("-i", "--image", required=True,

                help="path to image")

args = vars(ap.parse\_args())

nir = args['image']

#convert the rgb image to NIR

c.convert\_nir(args['image'])

#calculate the greeness of the image

crops\_ndvi\_a = nd.NDVI(nir, nir, False, False, args['image'])

crops\_data\_a = crops\_ndvi\_a.convert()

#calculate the greeness of the image

crops\_ndvi\_b = nd.NDVI\_B(nir, nir, False, False)

crops\_data\_b = crops\_ndvi\_b.convert()

#responsible for saving and loading images.

#calculate the wavelength for additional of greeness.

merge\_image = nd.merge(nir, nir, False, False)

resultImage = merge\_image.convert()

wavelength = merge\_image.run\_wavelength()

img1 = cv2.imread('image\_nir.png')

img2 = cv2.imread(args['image'])

img1 = imutils.resize(img1, width = 250)

img2 = imutils.resize(img2, width = 250)

h\_img = cv2.hconcat([img1, img2])

cv2.imwrite('combined\_raw.png', h\_img)

image = cv2.imread("NDVI.jpg")

crops\_image\_a = image[u.xx:(u.xx+u.xy), (u.yx):(u.yx+u.yy)]

crops\_image\_b = image[u.xx1:(u.xx1+u.xy1), (u.yx1):(u.yx1+u.yy1)]

dataw = {

        'leaf\_color\_a': crops\_data\_a['greeness'],

        'leaf\_color\_b': crops\_data\_b['greeness'],

        'leaf\_count\_a:': crops\_data\_a['leaf'],

        'leaf\_count\_b:': crops\_data\_b['leaf'],

        'canopy\_a' : crops\_data\_a['canopy'],

        'canopy\_b' : crops\_data\_b['canopy']

         }

crops\_identified\_a = cv2.imwrite('crops\_image\_a.png', crops\_image\_a)

crops\_identified\_b = cv2.imwrite('crops\_image\_b.png', crops\_image\_b)

# cv2.waitKey(0)

print(json.dumps(dataw))

#FOR MODEL EVALUATION

#After the completion of each training epoch, measure our performance on our test set.

#Put the model into evaluation mode

    model.eval()

# Reset the validation loss for this epoch.

    eval\_loss, eval\_accuracy = 0, 0

    nb\_eval\_steps, nb\_eval\_examples = 0, 0

    predictions , true\_labels = [], []

    for batch in valid\_dataloader:

        batch = tuple(t.to(device) for t in batch)

        b\_input\_ids, b\_input\_mask, b\_labels = batch

        with torch.no\_grad():

            outputs = model(b\_input\_ids, token\_type\_ids=None,

                            attention\_mask=b\_input\_mask, labels=b\_labels)

        logits = outputs[1].detach().cpu().numpy()

        label\_ids = b\_labels.to('cpu').numpy()

        eval\_loss += outputs[0].mean().item()

        predictions.extend([list(p) for p in np.argmax(logits, axis=2)])

        true\_labels.extend(label\_ids)

    eval\_loss = eval\_loss / len(valid\_dataloader)

    validation\_loss\_values.append(eval\_loss)

    print("Validation loss: {}".format(eval\_loss))

    pred\_tags = [tag\_values[p\_i] for p, l in zip(predictions, true\_labels)

                                 for p\_i, l\_i in zip(p, l) if tag\_values[l\_i] != "PAD"]

    valid\_tags = [tag\_values[l\_i] for l in true\_labels

                                  for l\_i in l if tag\_values[l\_i] != "PAD"]

    print("Validation Accuracy: {}".format(accuracy\_score(pred\_tags, valid\_tags)))

    print("Validation F1-Score: {}".format(accuracy\_score(pred\_tags, valid\_tags)))