Appendix 3: Python code for analyzing performance metric of downscaled images

This code is a module for the code in appendix 2

This code was called from a python notebook on a windows PC. Linux PCs seem to have an issue with the code.

```python
from cmath import sqrt
import numpy as np
import random
import matplotlib.pyplot as plt

def histogram_for_MCC(file_name, fasit_name, file_list, fasit_list, itterations,
hit_ratio, will_plot):


    # Defining RGB values for hit, miss, and out of bounds
    hit_rgb = (255,255,255,0)
    miss_rgb = (0,0,0,255)
    oob_rgb = (255,0,0,255)

    # Storing all filenames used from calling script
    filename = file_name
    fasit = fasit_name

    # Finding the location of all file parameters
    start_co   = filename.find('co')
    start_freq = filename.find('freq')
    start_illu = filename.find('Illu')
    start_Q    = filename.find('Q')
    start_zoom = filename.find('Zoom')
    start_case = filename.find('case')

    # Finding the values of all parameters
    surf = filename[start_Q:start_Q+2]
    co   = filename[start_co+3:start_co+6]
    freq = filename[start_freq+5:start_freq+7]
    illu = filename[start_illu+5:start_illu+7]
    zoom = filename[start_zoom-4:start_zoom-1]
    hit_sensitivity = str(hit_ratio*100) + '%'
    case = filename[start_case+5:start_case+7]

    # storing the image data of the RMS file and model file
    pixels3 = file_list
    pixels4 = fasit_list
```

```
fasit_binary = []
file_binary  = []

# Counting hits and misses model
for pixels in pixels4:
    if pixels == hit_rgb:
        fasit_binary.append(10)
    if pixels == miss_rgb:
        fasit_binary.append(0)

# Counting hits and misses RMS
for pixels in pixels3:
    if pixels == hit_rgb:
        file_binary.append(1)
    if pixels == miss_rgb:
        file_binary.append(0)

# Converting to arrays
fasit_array = np.array(fasit_binary)
file_array = np.array(file_binary)

# Object to randomize for permutation test
to_shuffle = file_binary

# Comparing hits and misses
observed = np.sum([fasit_array,file_array], 0)

# Counting confution matrix elements
t_p = np.count_nonzero(observed == 11)
t_n = np.count_nonzero(observed == 0)
f_p = np.count_nonzero(observed == 1)
f_n = np.count_nonzero(observed == 10)

# Preventing division-by-zero errors
if t_p+f_p == 0 or t_p+f_n == 0 or t_n+f_p == 0 or t_n+f_n == 0:
    MCC_observe = "Error"
    X5_plot = "Error"
    X95_plot = "Error"
    X5_MCC = "Error"
    X95_MCC = "Error"
    X5_sensitivity = "Error"
    X95_sensitivity = "Error"
    X5_specificity = "Error"
```

```
        X95_specificity = "Error"
        X5_error_rate = "Error"
        X95_error_rate = "Error"
        X5_accuracy = "Error"
        X95_accuracy = "Error"
        X5_precision = "Error"
        X95_precision = "Error"
        X5_false_positive_rate = "Error"
        X95_false_positive_rate = "Error"
        sensitivity_observe         = "Error"
        specificity_observe         = "Error"
        error_rate_observe          = "Error"
        accuracy_observe            = "Error"
        precision_observe           = "Error"
        false_positive_rate_observe = "Error"

    # If no division-by-zero error will occur the script continues
    else:

        # Performance metrics are calculated
        MCC_complex         = (t_p*t_n-
f_p*f_n)/sqrt((t_p+f_p)*(t_p+f_n)*(t_n+f_p)*(t_n+f_n))
        sensitivity_observe         = t_p / (t_p + f_p)
        specificity_observe         = t_n / (t_n + f_n)
        error_rate_observe          = (f_p + f_n) /(t_n+t_p+f_n+f_p)
        accuracy_observe            = (t_p + t_n) /(t_n+t_p+f_n+f_p)
        precision_observe           = t_p / (t_p + f_p)
        false_positive_rate_observe = f_p / (t_n + f_n)
        MCC_observe = MCC_complex.real
        observed_data = [t_p,t_n,f_p,f_n,MCC_observe]

        # Object for performance metrics of random RMS
        curve_data    = []

        # Permutation itterations fetched from calling file
        hist_number = itterations

        # Permutations performed
        for i in range(hist_number):

            # RMS data is randomized
            random.shuffle(to_shuffle)

            # random RMS data is converted to an array
            shuffle_array = np.array(to_shuffle)
```

```
            # random RMS data is compared to the model image (Actual case)
            test = np.sum([fasit_array,shuffle_array], 0)
            t_p = np.count_nonzero(test == 11)
            t_n = np.count_nonzero(test == 0)
            f_p = np.count_nonzero(test == 1)
            f_n = np.count_nonzero(test == 10)

            # The performance metrics of the random RMS are calculated
            MCC = (t_p*t_n-f_p*f_n)/sqrt((t_p+f_p)*(t_p+f_n)*(t_n+f_p)*(t_n+f_n))
            sensitivity         = t_p / (t_p + f_p)
            specificity         = t_n / (t_n + f_n)
            error_rate          = (f_p + f_n) /(t_n+t_p+f_n+f_p)
            accuracy            = (t_p + t_n) /(t_n+t_p+f_n+f_p)
            precision           = t_p / (t_p + f_p)
            false_positive_rate = f_p / (t_n + f_n)

            # The performance metrics are stored
            curve_data.append([t_p,t_n,f_p,f_n,MCC.real,sensitivity, specificity,
error_rate, accuracy, precision, false_positive_rate])

        # The performance metrics converted to an array
        curve_array = np.array(curve_data)

        # selecting which column to plot (4 is MCC)
        column = 4

        # Extracting plot data of random RMS
        x_plot = curve_array[:,column]

        # Extracting all other random data individually
        x_MCC = curve_array[:,4]
        x_sensitivity = curve_array[:,5]
        x_specificity = curve_array[:,6]
        x_error_rate = curve_array[:,7]
        x_accuracy = curve_array[:,8]
        x_precision = curve_array[:,9]
        x_false_positive_rate = curve_array[:,10]

        # Defining percentiles to calculate
        # (you should probably change this to 5 and 95)
        # (and not make the typo I made)
        lower_percentile = 0.5
        upper_percentile = 99.5
```

```
        # Calculating percentiles for all performance metrics
        X5_plot = np.percentile(x_plot, lower_percentile)
        X95_plot = np.percentile(x_plot, upper_percentile)
        X5_MCC = np.percentile(x_MCC, lower_percentile)
        X95_MCC = np.percentile(x_MCC, upper_percentile)
        X5_sensitivity = np.percentile(x_sensitivity, lower_percentile)
        X95_sensitivity = np.percentile(x_sensitivity, upper_percentile)
        X5_specificity = np.percentile(x_specificity, lower_percentile)
        X95_specificity = np.percentile(x_specificity, upper_percentile)
        X5_error_rate = np.percentile(x_error_rate, lower_percentile)
        X95_error_rate = np.percentile(x_error_rate, upper_percentile)
        X5_accuracy = np.percentile(x_accuracy, lower_percentile)
        X95_accuracy = np.percentile(x_accuracy, upper_percentile)
        X5_precision = np.percentile(x_precision, lower_percentile)
        X95_precision = np.percentile(x_precision, upper_percentile)
        X5_false_positive_rate = np.percentile(x_false_positive_rate,
lower_percentile)
        X95_false_positive_rate = np.percentile(x_false_positive_rate,
upper_percentile)

        # Checking if histograms should be made
        if will_plot == "Yes":

            # Objects for counting bins needed
            x_unique = []
            x_abs_unique = []

            # Counting unique values
            for i in x_plot:
                if i not in x_unique:
                    x_unique.append(i)
                    if abs(i) not in x_abs_unique:
                        x_abs_unique.append(i)

            # Defining range of histogram
            upper = max(x_abs_unique)
            lower = -max(x_abs_unique)
            buffer = 0 # 0.5* (upper - lower) / len(x_unique)

            # Creating histogram with title, labels and style
            plt.clf()
            plt.style.use('ggplot')
            data = plt.hist(x_plot, bins=len(x_unique), color = 'r',
edgecolor="black" ,range = [lower-buffer,upper+buffer])
```

```
            plt.title("Histogram for " +str(hist_number)+ " elements" + "\n" +
"Case: " + case + " Surface: " + surf + " Zoom: " + zoom + "x HT: " +
hit_sensitivity)
            plt.xlabel("MCC values")
            plt.ylabel("Frequency")


            # Creating X values for percentiles
            X5_real = [X5_plot,X5_plot]
            X95_real = [X95_plot,X95_plot]

            # Fetching observed MCC data
            X = [observed_data[column], observed_data[column]]

            # Creating Y axis values for percentiles and observed MCC
            Y = [0, max(data[0])]

            # Plotting percentiles and observed MCC
            plt.plot(X,Y, 'k', label = 'observed value')
            plt.plot(X5_real,Y, 'b:', label = str(lower_percentile) + 'th
percentile')
            plt.plot(X95_real,Y, 'r:', label = str(upper_percentile) + 'th
percentile')

            # Adding a legend
            plt.legend(loc = 'upper right')

            # Creating a png file of the complete plot
            plt.savefig("Results/" + filename + "_hist.png")

    # Returning desired data to calling file
    return [MCC_observe, X5_MCC, X95_MCC, sensitivity_observe, X5_sensitivity,
X95_sensitivity, specificity_observe, X5_specificity, X95_specificity,
error_rate_observe, X5_error_rate, X95_error_rate, accuracy_observe, X5_accuracy,
X95_accuracy, precision_observe, X5_precision, X95_precision,
false_positive_rate_observe, X5_false_positive_rate, X95_false_positive_rate]
```