

# IMAGE PROCESSING PROJECT - Acquadro Patrizio (502311), Uttini Lorenzo (502183)

## Preprocessing part

### Section 1: Import the Dataset and Create the Filepath

In the first section we start by importing the dataset, composed by the collect images, and we create the filepath that we will using during the project.

The function `dir` is used to get a list of all files in the directory specified by `path`.

```
% Define the path from th local machine
path = './Dataset_Final';

% Define the directory containing our dataset
dataset = dir(fullfile(path, '*.jpg'));
```

We can also extract the names of the images (filenames) inside the directory and construct an array containing the paths for each image through the `cellfun` function.

```
% Define the names
names_im = {dataset.name}';

% Create an array of paths
paths_im = cellfun(@(s) fullfile(path, s), names_im, 'UniformOutput',
false);

% Show the number of files (image) in the dataset
disp(numel(names_im))
```

138

We can check the first 5 images.

```
% Show the first and last 5 file names
disp(names_im(1:5))
```

```
{'01-1.jpg'}
{'02-1.jpg'}
{'03-2.jpg'}
{'04-3.jpg'}
{'05-1.jpg'}
```

### Section 2: Set the Classes (Ground Truth)

This section analyzes the dataset by extracting and counting the class labels associated with each image. The filenames contain the class information as the digit preceding the `.jpg` extension, since each filename is in the form `'n-c.jpg'`, where  $n$  is the number of the image (eg. 1; 10; 100) and  $c$  is the class (1; 2; 3)

The code uses regular expressions to extract this digit, converts with `str2double` the extracted class strings to numeric values, and compiles them into an array, called `target` through the `cell2mat` function.

```
% Extract the class index (ground truth) associated with each file
target = cellfun(@(s) regexp(s, '-(\d)\.jpg$', 'tokens'), names_im,
'UniformOutput', false); % regexp to extract class (number before .jpg)
target = cellfun(@(c) str2double(c{1}), target, 'UniformOutput', false); %
to convert the string in number
target = cell2mat(target); % to convert the number in a numerical array

% Show the first 5 classes in the target array
disp(target(1:5))
```

```
1
1
2
3
1
```

With the function `unique` we define how many different classes we have in the dataset:

```
% Count the unique classes
disp(numel(unique(target)))
```

```
3
```

As we can note, there are **3 different classes** in the dataset. Each class has a precise meaning:

- **1: Correctly Exposed** . The image is exposed with normal values, meaning that the brightness levels are balanced, and details are visible in both the shadows and highlights..
- **2: Underexposed**. The image is too dark, indicating that it did not receive enough light. This results in loss of detail in the shadow areas, making parts of the image appear too dark or completely black.
- **3: Overexposed**. The image is too bright, indicating that it received too much light. This results in loss of detail in the highlight areas, making parts of the image appear washed out or completely white.

Moreover, we can see how many instances we have for each class to understand if the dataset is well-balanced.

```
% Find the unique classes
classes = unique(target);

% Initialize an array to hold the count of images for each class
class_counts = zeros(size(classes));

% Loop over each class and count the occurrences
for i = 1:numel(classes)
    class = classes(i);
    class_counts(i) = sum(target == class);
end

% Show the number of images for each class
```

```
for i = 1:numel(classes)
    fprintf('Class %d: %d images\n', classes(i), class_counts(i));
end
```

```
Class 1: 46 images
Class 2: 46 images
Class 3: 46 images
```

The dataset contains a total of *138 images*, with *46 images per class*.

This **perfectly balanced distribution** ensures that each class is equally represented, which is crucial for unbiased training and evaluation in subsequent image processing and classification tasks. The balanced nature of the dataset helps in creating robust models that do not favor any particular class.

## Section 3: Preparing the Dataset for Gamma Correction

This section prepares the dataset for the gamma correction task by **selecting**, **pre-processing**, and **organizing** the images from each class.

First, the *number of images to process* is defined, which is set to 138 to utilize the entire dataset for robust results. This parameter can be adjusted to manage **computational resources**, particularly to *avoid crashes* on less powerful hardware.

```
% Set the number of images to process for each class
num_images_to_process = 46; % Adjust as needed to manage computational
resources
```

The **indices** of images belonging to each class (correctly exposed, underexposed, overexposed) are identified using the target array. The find function is used to locate the positions of images in each class:

- `correctly_exposed_indices` contains the indices of images that are correctly exposed (Class 1).
- `underexposed_indices` contains the indices of images that are underexposed (Class 2).
- `overexposed_indices` contains the indices of images that are overexposed (Class 3).

For each class, the *number of images to be processed* is determined by taking the minimum of `num_images_to_process` and the actual number of available images in that class. This ensures that we do not attempt to process more images than are present in the dataset. The images are selected in ascending order, meaning the first *n* images for each class.

```
% Select a subset of images for each class
correctly_exposed_indices = find(target == 1);
underexposed_indices = find(target == 2);
overexposed_indices = find(target == 3);

if ~isempty(correctly_exposed_indices)
    num_correctly_to_process = min(num_images_to_process,
length(correctly_exposed_indices));
    correctly_exposed_indices =
correctly_exposed_indices(1:num_correctly_to_process);
end
```

```

if ~isempty(underexposed_indices)
    num_underexposed_to_process = min(num_images_to_process,
length(underexposed_indices));
    underexposed_indices =
underexposed_indices(1:num_underexposed_to_process);
end

if ~isempty(overexposed_indices)
    num_overexposed_to_process = min(num_images_to_process,
length(overexposed_indices));
    overexposed_indices = overexposed_indices(1:num_overexposed_to_process);
end

```

Lists are *preallocated*, improving memory management and execution speed, to store the **images** and their corresponding **filenames** for each class, using the file paths. Filenames are saved since will be later used when we will save the images corrected with the gamma function.

Finally, the images are converted to **double precision** and transformed into the **YCbCr** color space. **Double precision** *normalizes* the image values within the range 0-1, which is essential for correctly applying gamma correction. The **YCbCr color space** separates image *luminance* (Y) from *chrominance* (Cb and Cr), leading to more effective gamma adjustments on the luminance channel without affecting the color information, so to more accurate and visually pleasing results.

```

% Preallocate lists for underexposed, overexposed, and correctly exposed
images
underexposed_images = cell(1, num_underexposed_to_process);
overexposed_images = cell(1, num_overexposed_to_process);
correctly_exposed_images = cell(1, num_correctly_to_process);

% Preallocate lists for underexposed, overexposed, and correctly exposed
filenames
underexposed_filenames = cell(1, num_underexposed_to_process);
overexposed_filenames = cell(1, num_overexposed_to_process);
correctly_exposed_filenames = cell(1, num_correctly_to_process);

% Populate the arrays with the corresponding images and filenames
for i = 1:num_correctly_to_process
    correctly_exposed_images{i} =
imread(paths_im{correctly_exposed_indices(i)});
    correctly_exposed_filenames{i} = paths_im{correctly_exposed_indices(i)};
end

for i = 1:num_underexposed_to_process
    underexposed_images{i} = imread(paths_im{underexposed_indices(i)});
    underexposed_filenames{i} = paths_im{underexposed_indices(i)};
end

for i = 1:num_overexposed_to_process
    overexposed_images{i} = imread(paths_im{overexposed_indices(i)});

```

```
overexposed_filenames{i} = paths_im{overexposed_indices(i)};
end
```

We **split** this parts of code in order **to avoid crush** of the program, by running it when the other parts before have finished to run.

```
% Pre-process underexposed images: convert to double and YCbCr
for i = 1:length(underexposed_images)
    underexposed_images{i} = im2double(rgb2ycbcr(underexposed_images{i}));
end
```

```
% Pre-process overexposed images: convert to double and YCbCr
for i = 1:length(overexposed_images)
    overexposed_images{i} = im2double(rgb2ycbcr(overexposed_images{i}));
end
```

```
% Convert the correctly exposed images to double and YCbCr
for i = 1:length(correctly_exposed_images)
    correctly_exposed_images{i} =
    im2double(rgb2ycbcr(correctly_exposed_images{i}));
end
```

## Classification Task

### Section 1: Extraction of the features

**Extracting features** such as mean, variance, and other statistical measures from images is crucial in image classification projects because it *reduces the high dimensionality* of raw image data, making it more manageable for machine learning algorithms. It captures essential and *relevant information* like brightness, contrast, and texture, which helps in distinguishing between different classes. Additionally, using well-chosen features *enhances the accuracy and robustness* of machine learning models, and makes the decision-making process more understandable and interpretable by focusing on a smaller set of meaningful features.

The functions to extract the features are defined in separated mini-scripts (**external functions**) that are called with the special symbol "@" used to define handle functions in MATLAB. To extract the features, we will use just the training set defined earlier.

Before extracting our features, we can define a general function handle, called `extract_feat`, to pass a specific feature extractor to a generic function that processes the images. This approach makes the code reusable and adaptable for different feature extraction methods. We also define a list called `features` where we store all the features that we extract.

```
features = [];
```

### Feature Set 1: First 3 moments on different colorspace

This feature set involves extracting the first three statistical moments (*mean*, *variance*, and *skewness*) for each channel of an image in different color spaces (*RGB*, *HSV*, and *YCbCr*). These moments provide essential information about the distribution of pixel intensities, which is crucial for classifying images based on their exposure levels.

### The First Three Moments

1. **Mean (First Moment):** The mean is a measure of central tendency, representing the average value of the pixel intensities in an image channel. In the context of exposure classification, the mean provides a straightforward indication of the overall brightness or color intensity of the image.
2. **Variance (Second Moment):** Variance measures the dispersion or spread of the pixel intensities around the mean. It quantifies how much the intensity values deviate from the average intensity. In exposure classification, variance is crucial because it gives insights into the contrast of the image.
3. **Skewness (Third Moment):** Skewness measures the asymmetry of the intensity distribution around the mean. It indicates whether the distribution of pixel intensities is biased towards the higher end (positive skewness) or the lower end (negative skewness).

### Color Spaces

1. **RGB Color Space:** The RGB color space represents images using three color channels: *Red*, *Green*, and *Blue*. It is an additive color space, meaning colors are created by combining different intensities of these three primary colors. Each pixel in an RGB image is defined by a triplet of values corresponding to the intensities of red, green, and blue.
2. **HSV Color Space:** The HSV color space represents images using three components: *Hue*, *Saturation*, and *Value*. Hue describes the type of color (red, green, blue, etc.), saturation describes the intensity or purity of the color, and value describes the brightness of the color. This color space is often used in color analysis because it separates the color information (hue) from the intensity information (value).
3. **YCbCr Color Space:** The YCbCr color space separates image luminance from chrominance, with Y representing the *luminance* (brightness), Cb representing the *blue-difference* chroma component, and Cr representing the *red-difference* chroma component. This color space is widely used in video compression and digital imaging.

Each moment and each colorspace gives an important piece of information to complete our classification task. They describe different aspects of the images, and employed all together can be really efficient to recognize the exposure of an image. For instance, a higher mean value in the RGB channels could indicate a brighter image, which might be overexposed, while a lower mean value could indicate an underexposed image. At the same time, high variance may indicate significant differences in pixel intensities, which is often characteristic of correctly exposed images with good contrast. The same approach could be used for the colorspace where, for example, a high mean value in all three RGB channels may indicate an overexposed image and a high mean value in the V channel (HSV space) suggests a bright image, potentially overexposed.

However, it's not always straightforward. *Noise* and *distortions* can affect pixel intensity distributions, making it difficult to rely on a single color space or moment for accurate classification. Therefore, combining the first three moments (mean, variance, skewness) from multiple color spaces (RGB, HSV, YCbCr) captures a comprehensive view of the image's exposure characteristics.

## RGB

```
% Extract the first 3 moments for RGB
```

```
features_moments_rgb = extract_feat(paths_im, @feat_moments_rgb);
```

```
20/138 (14.49%)  
40/138 (28.99%)  
60/138 (43.48%)  
80/138 (57.97%)  
100/138 (72.46%)  
120/138 (86.96%)
```

```
% Dimension of rgb
```

```
disp(size(features_moments_rgb));
```

```
138    9
```

```
% Example of the first 2 rows
```

```
disp(features_moments_rgb(1:2,:));
```

```
    0.6503    0.0401   -0.8802    0.4782    0.0515   -0.2195    0.4097    0.0610    0.2463  
    0.4829    0.0587    0.1515    0.5448    0.0597   -0.3108    0.6154    0.0834   -0.8047
```

```
% Save the features to a .mat file for a possible reuse
```

```
save('features_moments_rgb.mat', 'features_moments_rgb');
```

## HSV

```
% Extract the first 3 moments for HSV
```

```
features_moments_hsv = extract_feat(paths_im, @feat_moments_hsv);
```

```
20/138 (14.49%)  
40/138 (28.99%)  
60/138 (43.48%)  
80/138 (57.97%)  
100/138 (72.46%)  
120/138 (86.96%)
```

```
% Dimension of hsv
```

```
disp(size(features_moments_hsv));
```

```
138    9
```

```
% Example of the first 2 rows
```

```
disp(features_moments_hsv(1:2,:));
```

```
    0.3668    0.1602    0.6571    0.4151    0.0872    0.3535    0.6578    0.0366   -0.7492  
    0.4390    0.0407   -0.5911    0.2983    0.0479    0.6880    0.6418    0.0723   -0.8936
```

```
% Save the features to a .mat file for a possible reuse
```

```
save('features_moments_hsv.mat', 'features_moments_hsv');
```

## YCbCr

```
% Extract the first 3 moments for YCbCr
```

```
features_moments_ycbcr = extract_feat(paths_im, @feat_moments_ycbcr);
```

```
20/138 (14.49%)
40/138 (28.99%)
60/138 (43.48%)
80/138 (57.97%)
100/138 (72.46%)
120/138 (86.96%)
```

```
% Dimension of YCbCr
```

```
disp(size(features_moments_ycbcr));
```

```
138    9
```

```
% Example of the first 2 rows
```

```
disp(features_moments_ycbcr(1:2,:));
```

```
0.5109    0.0313   -0.2684    0.4463    0.0044   -0.6954    0.5824    0.0049    0.5534
0.5216    0.0435   -0.2667    0.5422    0.0040    0.3355    0.4697    0.0016    0.1418
```

```
% Save the features to a .mat file for a possible reuse
```

```
save('features_moments_ycbcr.mat', 'features_moments_ycbcr');
```

As we can note each colorspace has 9 features (3 for each channel). In total, the first set of features has **27 features**.

## Feature Set 2: Histograms

This feature set includes two types of histograms: *pixel intensity distributions* in RGB channels and *edge direction distributions* in RGB channels. These histograms provide detailed information about color and structural characteristics essential for classifying exposure levels.

### Distribution of Intensities in RGB Channels

This histogram captures the distribution of *pixel intensities* for the Red, Green, and Blue (RGB) channels. Each channel's intensity values (0 to 255) are divided into 5 bins, counting the pixels in each bin. The histograms for the three channels are then concatenated into a 15-dimensional feature vector.

The intensity histograms reveal the color distribution in the image. Overexposed images could have higher counts in the brighter bins, while underexposed images in the darker bins.

```
% Define the feature for color histogram
```

```
features_inthistogram = extract_feat(paths_im, @feat_inthistogram);
```

```
20/138 (14.49%)
40/138 (28.99%)
60/138 (43.48%)
80/138 (57.97%)
100/138 (72.46%)
120/138 (86.96%)
```

```
% Dimensions of histogram
```

```
disp(size(features_inthistogram));
```

```
138    15
```



```
% Example of the first 2 rows
disp(features_inthistogram(1:2,:))
```

347690	502818	2126591	3207650	2109651	1368455	1573913	2608435	21851
1392889	3835403	3052297	2093097	1819082	1383592	1629145	4358265	26477

```
% Save the features to a .mat file for a possible reuse
save('features_inthistogram.mat', 'features_inthistogram');
```

## Edge Direction Histograms in RGB Channels

This histogram captures the distribution of *edge directions* for the RGB channels. For each channel, gradient magnitudes and directions are computed, and the directions are divided into 8 bins to create histograms. The histograms for the three channels are concatenated into a 24-dimensional feature vector.

Edge direction histograms highlight the structural characteristics of the image. Correctly exposed images typically have well-defined, diverse edges, while underexposed images may lack clear edges.

```
% Define the feature for edge histogram
features_edgehistogram = extract_feat(paths_im, @feat_edgehistogram);
```

```
20/138 (14.49%)
40/138 (28.99%)
60/138 (43.48%)
80/138 (57.97%)
100/138 (72.46%)
120/138 (86.96%)
```

```
% Dimension of edge histogram
disp(size(features_edgehistogram))
```

```
138    24
```

```
% Example of the first 2 rows
disp(features_edgehistogram(1:2,:))
```

764882	1000145	1058827	1154300	1171226	1021906	1094285	1028829	7591
1626042	1250887	1569450	977055	2968961	1387429	1620332	792612	14728

```
% Save the features to a .mat file for a possible reuse
save('features_edgehistogram.mat', 'features_edgehistogram');
```

As we can note, this set of features has 15 features from the first histogram and 24 features from the second histogram. Thus it has a total of **39 features**.

## Feature Set 3: Entropy

Entropy is a measure of *randomness* or complexity in an image. It quantifies the amount of information or detail present, making it a useful feature for analyzing image exposure. Usually, underexposed images tend to have lower entropy due to the lack of detail in dark areas, while overexposed images may also have lower entropy due to saturation in bright areas. Correctly exposed images, with balanced details and contrast, generally have higher entropy.

We calculate entropy for each RGB channel.

```
% Extract entropy features
features_entropy = extract_feat(paths_im, @feat_entropy);
```

```
20/138 (14.49%)
40/138 (28.99%)
60/138 (43.48%)
80/138 (57.97%)
100/138 (72.46%)
120/138 (86.96%)
```

```
% Dimension of entropy features
disp(size(features_entropy));
```

```
138      3
```

```
% Example of the first 2 rows
disp(features_entropy(1:2,:));
```

```
7.5189    7.7122    7.6663
7.7254    7.7668    7.4933
```

```
% Save the features to a .mat file for a possible reuse
save('features_entropy.mat', 'features_entropy');
```

As we can note, in this set we have just **3 features**.

## Combine all the set features

```
features = [features_moments_rgb, features_moments_hsv,
features_moments_ycbcr, features_inthistogram, features_edgehistogram,
features_entropy];
disp(size(features));
```

```
138      69
138      27
138      66
```

The total number of features is **69**.

## Section 2: Divide the dataset into training and test sets

To understand how the classifier performs on unseen images (new data), we have to divide the dataset into **training** and **test** sets. The training part (around 90% out of the total) will be used to train different classifiers through KFold cross validation. The test part (around 10% of the total) will be employed to understand how the best classifier, identified with the training process, performs with completely new data.

We define a separated function called `divide_dataset` to perform the splitting between train and test sets. The test set will contain just only 5 instances per class (around 10% of the total)

```
[trainFeatures, testFeatures, trainTarget, ~] = divide_dataset(features,
target);
```

```
% Show dimension of Train set
disp(size(trainFeatures))
```

123     69

```
% Show dimension of Test set
disp(size(testFeatures))
```

15     69

We also **normalize** all the features using just the training elements.

```
% Compute the mean and standard deviation of the training data
meanTrain = mean(trainFeatures);
stdTrain = std(trainFeatures);

% Normalize the training data
trainFeatures = (trainFeatures - meanTrain) ./ stdTrain;

% Normalize the test data using the training data statistics
testFeatures = (testFeatures - meanTrain) ./ stdTrain;
```

## Section 3: Train the classifiers

*Classification* in the image domain refers to the process of assigning a label to an image from a predefined set of categories. In this context, the categories are the exposure levels: **correctly exposed**, **underexposed**, and **overexposed**. Classification is a fundamental task in computer vision and image processing, with applications ranging from object detection to medical imaging.

To train the various classifiers, as KNN, SVM, Decision Trees, we use the training set of features (`trainFeatures`). During the training, we employ a **KFold Cross-Validation** (built-in method in the Classification Learner App) that divides the training dataset in *k folds*, and it uses one fold at time as validation set. This technique is really useful as it ensures that the evaluation is robust and not dependent on a specific split of the dataset.

On the Classification Learner we try all the possible classifiers and we get the following results as average accuracy on the validation sets:

(we show just the first 6 results)

```
% Specify the path
imagePath = 'performance_classifiers.jpg';

% Load the image
im = imread(imagePath);

% Show the image
figure, imshow(im), title('Performances of the classifiers');
```

## Performances of the classifiers

Models		
Sort by	Accuracy (Va...	↓ ↑
☆ 2.3	Tree	Accuracy (Validation): 82.9%
Last change: Coarse Tree 69/69 features		
☆ 2.23	Ensemble	Accuracy (Validation): 82.9%
Last change: Bagged Trees 69/69 features		
☆ 2.8	Naive Bayes	Accuracy (Validation): 82.1%
Last change: Gaussian Naive Bayes 69/69 features		
☆ 1	Tree	Accuracy (Validation): 81.3%
Last change: Fine Tree 69/69 features		
☆ 2.1	Tree	Accuracy (Validation): 81.3%
Last change: Fine Tree 69/69 features		
☆ 2.2	Tree	Accuracy (Validation): 81.3%
Last change: Medium Tree 69/69 features		
☆ 2.33	Kernel	Accuracy (Validation): 79.7%
Last change: Logistic Regression Kernel 69/69 features		

As we can note, we obtain the best accuracy (**82.9%**) with 3 classifiers:

- *Coarse Tree (Tree)*
- *Bagged Trees (Ensemble)*

As sanity check operation, we export the models here and retrain them with the same data

```
% Coarse Tree accuracy
[trainedTree, validationAccuracy] = coarseTree(trainFeatures, trainTarget);
disp(validationAccuracy)
```

0.7561

```
% Bagged Trees accuracy
[trainedBagged, validationAccuracy] = baggedTrees2(trainFeatures,
trainTarget);
disp(validationAccuracy)
```

0.8374

The results are really close to the original one.

In general, the performances of these three models are quite good. However, they are *not excellent performances* (>90% of accuracy) and the reason of that could be related to the dataset and the extracted features. In a Machine Learning project, as some papers have proven, the number of instances (training samples, in our case the image in the training set) should be at least 10 times the number of features. In our case, this ratio is not respected as the training instances are 123 and the features extracted are 69. The resulting ratio between training instances and features is less than 2!

Thus, there are possible solutions to obtain better performances:

- Collect more images
- Reduce the number of features

As the first option is more computationally expensive, we can try to reduce the number of features. Considering that we have 123 training instances, to respect the ratio of 10, we can select at most 12 features.

```
% Access feature importance
importance = predictorImportance(trainedBagged.ClassificationEnsemble);
```

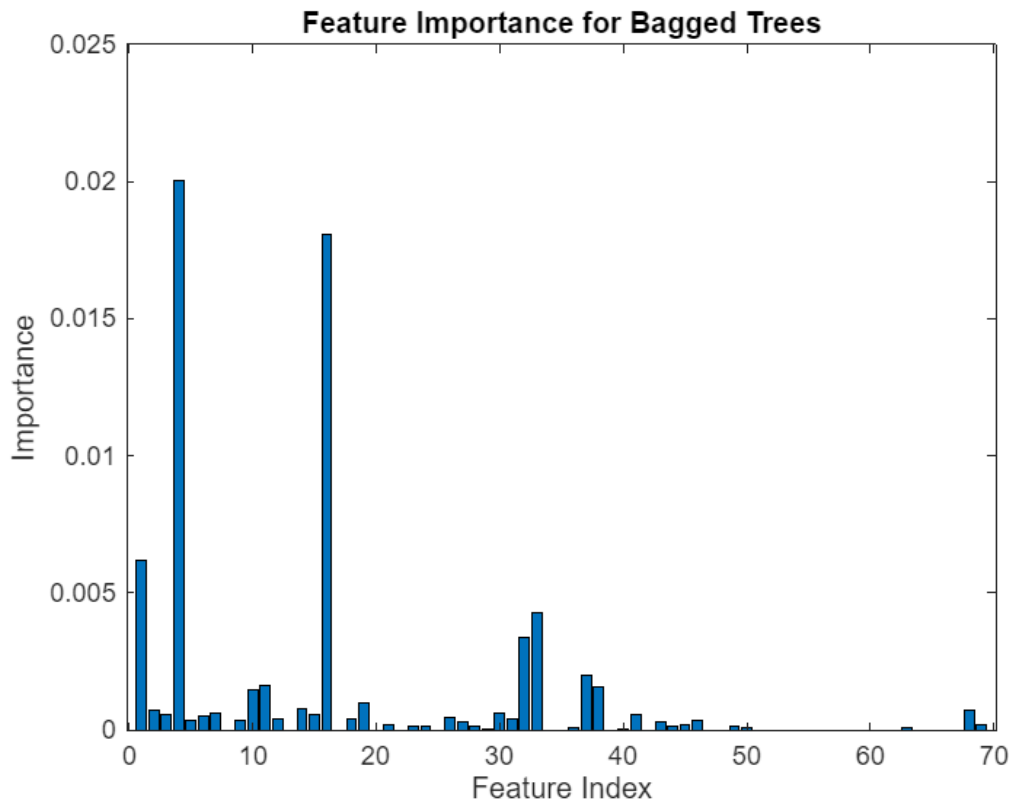
```
% Display the feature importance
disp('Feature Importance:');
```

Feature Importance:

```
disp(importance);
```

0.0062    0.0007    0.0006    0.0200    0.0003    0.0005    0.0006    0    0.0003    0.0015    0

```
% Visualize the feature importance
figure;
bar(importance);
title('Feature Importance for Bagged Trees');
xlabel('Feature Index');
ylabel('Importance');
```



We select the **10 most important features** and we describe them.

```
% Select the ten most important features
[~, sortedIdx] = sort(importance, 'descend');
topFeatures = sortedIdx(1:10);

% Display the top features
disp('Top 10 Features:');
```

Top 10 Features:

```
disp(topFeatures);
```

4 16 1 33 32 37 11 38 10 19

The top 10 features are:

- 1: *First moment (mean) for R channel in the RGB colorspace*
- 4: *First moment (mean) for G channel in the RGB colorspace*
- 10: *First moment (mean) for H channel in HSV colorspace*
- 11: *Second moment (variance) for H channel in HSV colorspace*
- 16: *First moment (mean) for V channel in HSV colorspace*
- 19: *First moment (mean) for Y channel in YCbCr colorspace*
- 32: *Intensity bin for R channel in RGB histogram*

- 33, 37: *Intensities bins for G channel* in RGB histogram
- 38: *Intensity bin for B channel* in RGB histogram

Thus, the most discriminative features in our models are basically the **first moments (means)** of almost any channels in different colorspace. Moreover, the **intensity histogram in RGB** includes some features that are fundamentals.

Any of the features extracted with edge histogram and with entropy are potentially useful for our task.

Now we can retrain the classifiers with only these set of features that we call `topFeatures`.

```
% Indices
indices = [4, 16, 1, 33, 32, 37, 11, 38, 10, 19];

% Select the specified features from the overall feature matrix and divide
% the dataset
topFeatures = features(:, indices);
[trainTopFeatures, testTopFeatures, trainTarget, testTarget] =
divide_dataset(topFeatures, target);

% Compute the mean and standard deviation of the training data
meanTrain = mean(trainTopFeatures);
stdTrain = std(testTopFeatures);
% Normalize the training data
trainTopFeatures = (trainTopFeatures - meanTrain) ./ stdTrain;
% Normalize the test data using the training data statistics
testTopFeatures = (testTopFeatures - meanTrain) ./ stdTrain;

% Show size
disp(size(trainTopFeatures))
```

```
123    10
```






**Results** of the performances are:

```
% Specify the path
imagePath = 'performance_selecfeat.jpg';

% Load the image
im = imread(imagePath);

% Show the image
figure, imshow(im), title('Performances of the classifiers with selected
features');
```

## Performances of the classifiers with selected features

Models		
Sort by Accuracy (Va...     		
☆ 2.7 Efficient ...	Accuracy (Validation): 90.2%	
Last change: Efficient Linear SVM 10/10 features		
☆ 2.10 SVM	Accuracy (Validation): 87.8%	
Last change: Linear SVM 10/10 features		
☆ 2.12 SVM	Accuracy (Validation): 86.2%	
Last change: Cubic SVM 10/10 features		
☆ 2.33 Kernel	Accuracy (Validation): 86.2%	
Last change: Logistic Regression Kernel 10/10 feat		
☆ 2.8 Naive Ba...	Accuracy (Validation): 85.4%	
Last change: Gaussian Naive Bayes 10/10 features		
☆ 2.21 KNN	Accuracy (Validation): 85.4%	
Last change: Weighted KNN 10/10 features		

As we can note, the *performance are definitely **increased***. The reduction of dimensionality (from 69 to 10 features) leads to a better performances for most part of the models. We export the **Efficient Linear SVM** with the best performance (**90.2%** of Accuracy on the validation set) and we retrain the model for a sanity check.

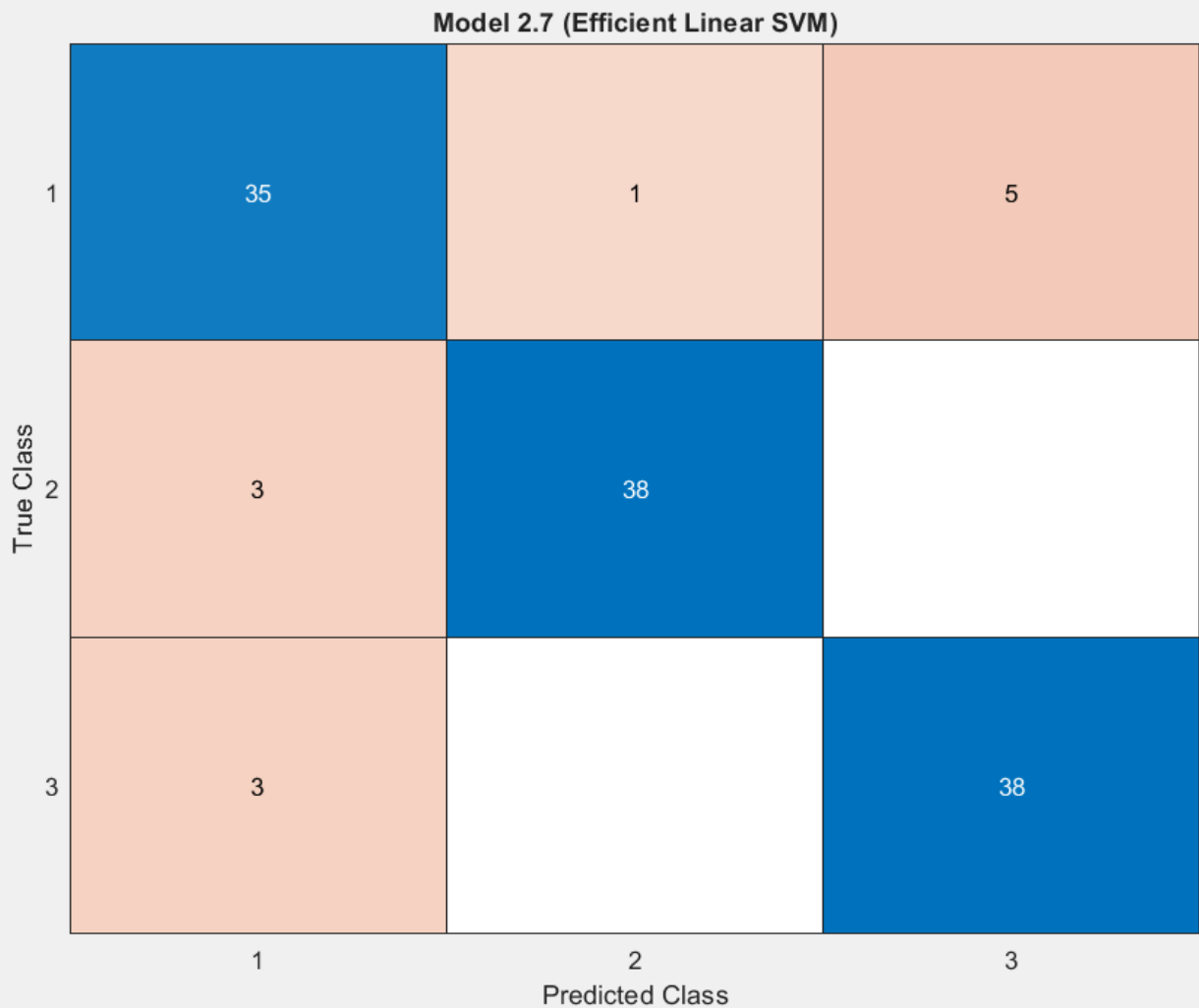
```
% Efficient Linear SVM accuracy
[trainedSVM, validationAccuracy] = effLinSVM(trainTopFeatures, trainTarget);
disp(validationAccuracy)
```

0.8211

We show also the **Confusion Matrix** to understand better how the classification worked:

```
% Show the Confusion Matrix
openfig("conf_matrix_train.fig", "visible");
```





The confusion matrix indicates that the **classification between class 2 (Underexposed) and class 3 (Overexposed) is perfect**. However, there are some errors in distinguishing between class 1 (Correctly Exposed) and class 2 (Underexposed), as well as between class 1 (Correctly Exposed) and class 3 (Overexposed). This outcome is expected, as the distinction between Underexposed and Overexposed images is more straightforward, whereas *Correctly Exposed images can sometimes be similar to either Underexposed or Overexposed images*.

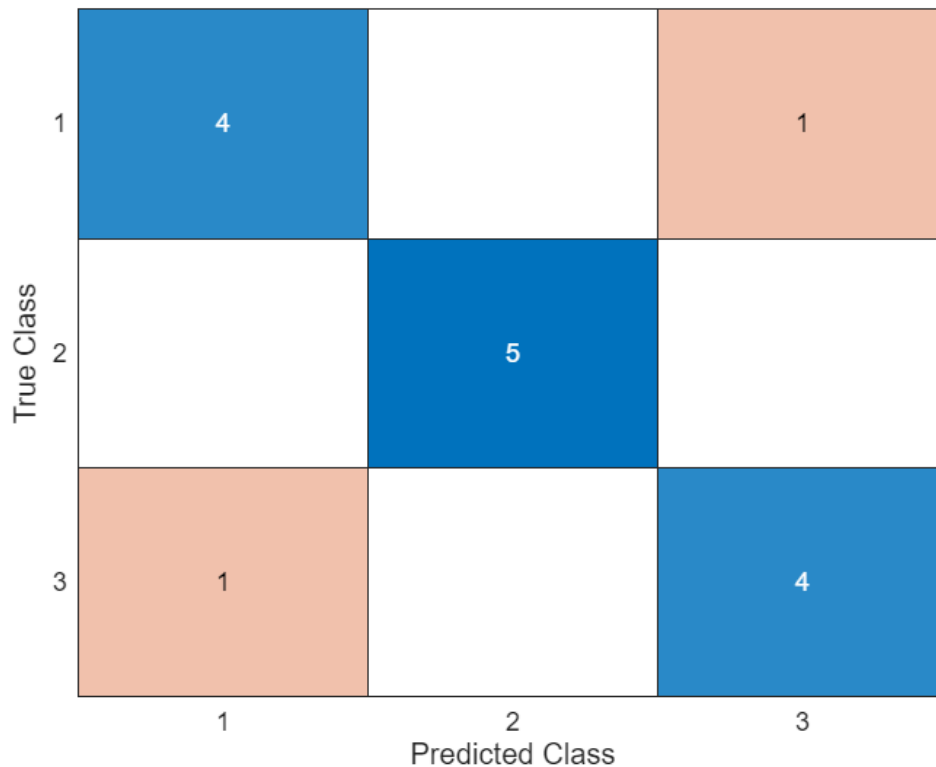
## Section 4: Evaluation of the best classifier

In this last section of the Classification Task, we evaluate on unseen data (Test data) the best classifier we obtain from the training. Our best classifier is the **Efficient Linear SVM** and we calculate the accuracy, recall, precision and f1 on the new instances (testTopFeatures).

```
% Predict on test data using the trained classifier
test_predictions = trainedSVM.predictFcn(testTopFeatures);

% Confusion matrix
```

```
conf_matrix = confusionmat(testTarget, test_predictions);
figure, confusionchart(conf_matrix)
```



```
% Precision, Recall, F1 Score
```

```
precision = diag(conf_matrix) ./ sum(conf_matrix, 2);
recall = diag(conf_matrix) ./ sum(conf_matrix, 1)';
f1 = 2 * (precision .* recall) ./ (precision + recall);
```

```
disp('Precision for each class:'), disp(precision);
```

```
Precision for each class:
    0.8000    1.0000    0.8000
```

```
disp('Recall for each class:'), disp(recall);
```

```
Recall for each class:
    0.8000    1.0000    0.8000
```

```
disp('F1 Score for each class:'), disp(f1);
```

```
F1 Score for each class:
    0.8000    1.0000    0.8000
```

```
% Overall Accuracy, Precision, Recall, F1 on the test set
```

```
overallAccuracy = sum(test_predictions == testTarget) / numel(testTarget);
overallPrecision = mean(precision);
overallRecall = mean(recall);
```

```

overallF1 = mean(f1);

disp(['Overall Accuracy: ', num2str(overallAccuracy)]);

Overall Accuracy: 0.86667

disp(['Overall Precision: ', num2str(overallPrecision)]);

Overall Precision: 0.86667

disp(['Overall Recall: ', num2str(overallRecall)]);

Overall Recall: 0.86667

disp(['Overall F1 Score: ', num2str(overallF1)]);

Overall F1 Score: 0.86667

```

The test performances are slightly worse than the training one (**86%** vs 90% accuracy). This is due to the small number of instances in the test set (5 for each class, so 15 in total). However, looking at the Confusion matrix we can state the final model misclassify just 2 instances.

We can conclude that the model we have built is **robust** and performs quite well on unseen data.

## SAVE MODEL

```

% We save the model for further reuse
save('trainedSVM.mat', 'trainedSVM', 'validationAccuracy');

```

# Gamma Correction Task

## Section 1: Mean and Variance of Luminance from Correctly Exposed Images

In this section, we calculate the **mean** and **variance** of the luminance (Y channel) from correctly exposed images. By accurately characterizing the luminance of correctly exposed images, we can determine the appropriate adjustments needed to correct underexposed and overexposed images, ensuring that they match the target luminance characteristics. This step is essential for achieving consistent and visually pleasing results across the entire dataset.

The luminance channel (Y) is extracted from each image, and its mean and variance are computed. These values are stored in *arrays* `norm_means` and `norm_vars`.

Next, the **average** of these means and variances is calculated to obtain a single representative mean and variance for the entire dataset of correctly exposed images. This representative mean (`target_mean`) and variance (`target_variance`) are printed for verification.

```

% Calculate target mean and variance from correctly exposed images
norm_means = zeros(size(correctly_exposed_images));
norm_vars = zeros(size(correctly_exposed_images));

for i = 1:numel(correctly_exposed_images)

```

```

Y_channel = correctly_exposed_images{i}(:, :, 1);
norm_means(i) = mean(Y_channel, 'all');
norm_vars(i) = var(Y_channel, 0, 'all');
end

```

```

% Calculate the mean of the means and variances
target_mean = mean(norm_means);
target_variance = mean(norm_vars);

```

```

% Print the final target mean
fprintf('Target Mean: %f\n', target_mean)

```

Target Mean: 0.473533

```

% Print the final target variance
fprintf('Target Variance: %f\n', target_variance)

```

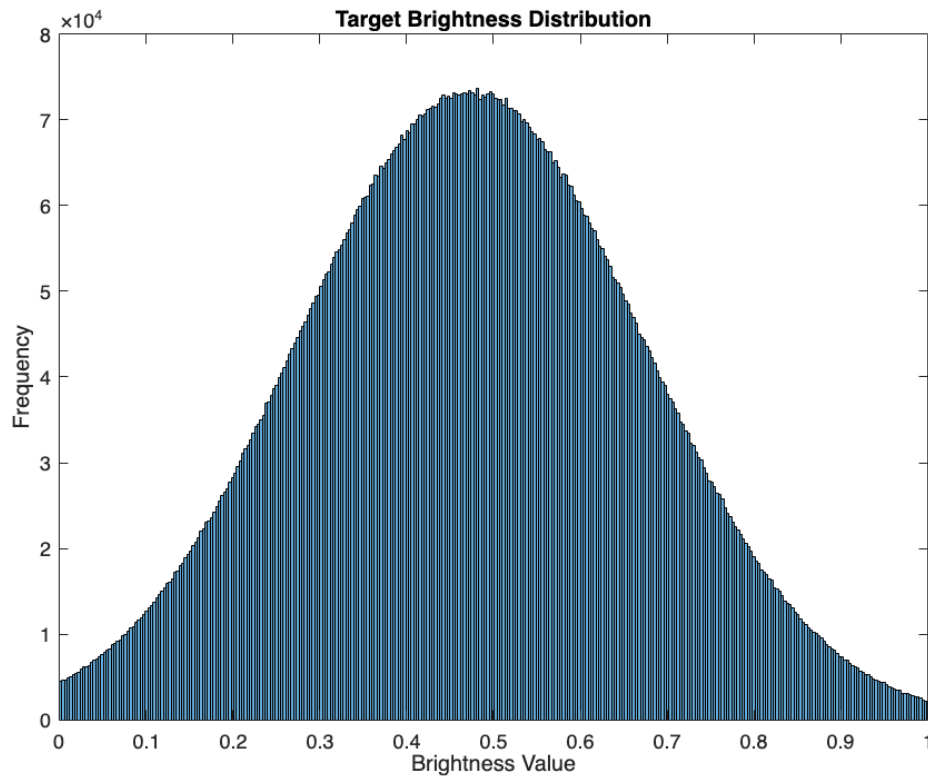
Target Variance: 0.039633

To visualize the **distribution** of the target luminance, a normal distribution is approximated using the calculated *target mean* and *target variance*.

```

% Visualize target brightness distribution
% NOTE that images are almost never normally distributed,
% We use a normal approximation just for the sake of visualization
target_dist = makedist("Normal", "mu", target_mean, "sigma",
sqrt(target_variance));
target_lum = random(target_dist, size(Y_channel));
figure, histogram(target_lum), xlim([0 1]), title('Target Brightness
Distribution'), xlabel('Brightness Value'),
ylabel('Frequency');

```



## Section 2 - Gamma Correction Method from the Lectures

In this section, we apply gamma correction to underexposed and overexposed images using **fixed gamma values**. This method involves setting predetermined gamma values for underexposed and overexposed images and then applying these corrections to randomly selected images from each class.

**Setting Gamma Values:** Two fixed gamma values are defined:

- `gamma_value_underexposed = 0.7` for underexposed images
- `gamma_value_overexposed = 1.3` for overexposed images

These values are chosen because gamma values *less than 1* are used to brighten underexposed images, while gamma values *greater than 1* are used to darken overexposed images. This choice is based on the understanding that gamma correction can effectively adjust the brightness of images to improve their exposure levels.

**Selecting Images:** The number of images to process is set to 1 (`num_images_to_process = 1`), but this value can be adjusted based on the user's *hardware* capabilities and *performance* needs. Random indices are selected from the available underexposed and overexposed images using the `randperm` function. This random selection ensures that different images are processed each time the code is run, *preventing monotony* and providing a diverse set of results for analysis.

**Applying Gamma Correction:** The gamma correction is applied to the selected images using the custom function `gammaCorrectionYCbCr`, defined in a separate script. This function extracts the Y, Cb, and Cr

channels, applies the gamma correction to the Y channel, and then recombines the corrected Y channel with the original Cb and Cr channels to form the corrected YCbCr image.

**Visualizing Results:** For each corrected image, a figure is then created with two subplots:

- The original underexposed or overexposed image
- The gamma-corrected image

```
% Set the gamma values for correction
gamma_value_underexposed = 0.7; % Example gamma value for underexposed
images
gamma_value_overexposed = 1.3; % Example gamma value for overexposed images

% Number of images to apply gamma correction to (change as needed)
num_images_to_process = 1;

% Select random images from each class
if ~isempty(underexposed_images)
    underexposed_indices = randperm(length(underexposed_images),
num_images_to_process);
end

if ~isempty(overexposed_images)
    overexposed_indices = randperm(length(overexposed_images),
num_images_to_process);
end

% Process random underexposed images
for i = 1:num_images_to_process
    if ~isempty(underexposed_images)
        original_underexposed_image =
underexposed_images{underexposed_indices(i)};
        corrected_underexposed_image =
gammaCorrectionYCbCr(original_underexposed_image, gamma_value_underexposed);

        % Visualize original and corrected underexposed image
        figure;
        subplot(1, 2, 1);
        imshow(ycbcr2rgb(original_underexposed_image));
        title('Original Underexposed');

        subplot(1, 2, 2);
        imshow(ycbcr2rgb(corrected_underexposed_image));
        title(['Gamma Corrected (gamma = ',
num2str(gamma_value_underexposed), ')']);
    end
end
```

Original Underexposed



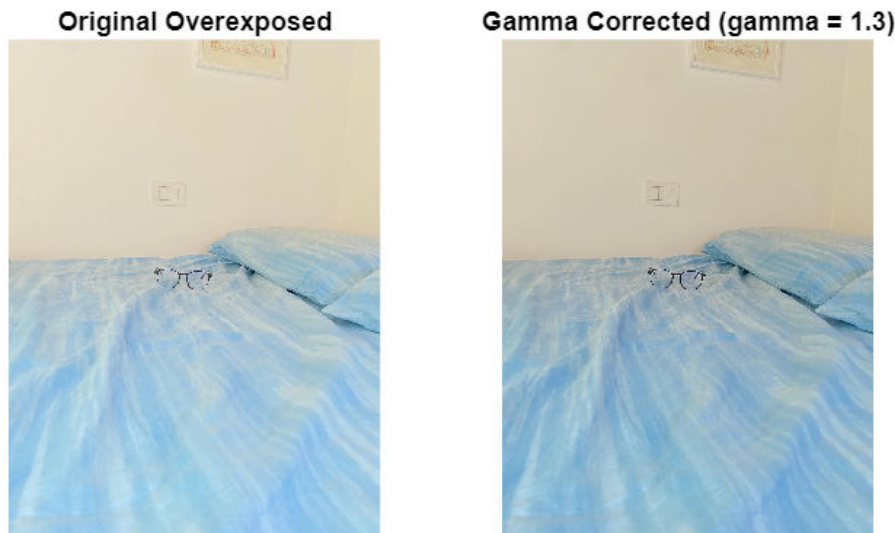
Gamma Corrected (gamma = 0.7)



```
% Process random overexposed images
for i = 1:num_images_to_process
    if ~isempty(overexposed_images)
        original_overexposed_image =
overexposed_images{overexposed_indices(i)};
        corrected_overexposed_image =
gammaCorrectionYCbCr(original_overexposed_image, gamma_value_overexposed);

        % Visualize original and corrected overexposed image
        figure;
        subplot(1, 2, 1);
        imshow(ycbcr2rgb(original_overexposed_image));
        title('Original Overexposed');

        subplot(1, 2, 2);
        imshow(ycbcr2rgb(corrected_overexposed_image));
        title(['Gamma Corrected (gamma = ',
num2str(gamma_value_overexposed), ')']);
    end
end
```



### Section 3 - Brute Force Method

In this section, we apply gamma correction to underexposed, overexposed, and correctly exposed images using a **brute force** method based on **RMSE** (Root Mean Square Error). This method involves testing a range of gamma values for each image to find the optimal gamma that minimizes the RMSE between the corrected image's luminance statistics and the target mean and variance.

**Setting Gamma Values:** A range of gamma values from 0.1 to 3.0 in increments of 0.1 is defined to test for all images. This range is broad enough to cover both brightening and darkening adjustments needed for underexposed and overexposed images, respectively.

**Applying Gamma Correction:** The gamma correction is applied to the selected images using the custom function `find_best_gamma`, defined in a separate script. This function implements a brute force method, which systematically tests each gamma value, calculates the corrected image's luminance statistics, and computes the RMSE for these statistics compared to the target values. The gamma value that results in the lowest RMSE is selected as the best gamma for correcting the image.

More in details, the logic behind this method involves the following steps, for each gamma value:

1. The function `find_best_gamma` calls `gammaCorrectionYCbCr` to apply each gamma value to the Y channel of the image and obtain the respective gamma-corrected images.
2. For each gamma-corrected image, `evaluate_exposure` calculates the RMSE based on the differences between the corrected images mean and variance and the target values, that we found before (Section 1).



3. The gamma value with the lowest RMSE is identified as the best gamma, and the corresponding corrected image is selected.

The RMSE (Root Mean Square Error) is a metric calculated as the square root of the mean of the squared differences between observed and predicted values. It is used for gamma correction because it provides a single measure of accuracy that combines the differences between predicted and observed values (variance and mean). It is particularly useful for quantifying the distance between the luminance statistics of the corrected image and the target values, penalizing large deviations and resulting in a more precise correction.

```
% Parameters
gamma_values = 0.1:0.1:3.0; % Single range of gamma values to test for all
images

% Select two random images from each class
num_images_to_process = 2;

if ~isempty(underexposed_images)
    underexposed_indices = randperm(length(underexposed_images),
num_images_to_process);
end

if ~isempty(overexposed_images)
    overexposed_indices = randperm(length(overexposed_images),
num_images_to_process);
end

if ~isempty(correctly_exposed_images)
    correctly_exposed_indices = randperm(length(correctly_exposed_images),
num_images_to_process);
end

% Process random underexposed images
for i = 1:num_images_to_process
    if ~isempty(underexposed_images)
        original_underexposed_image =
underexposed_images{underexposed_indices(i)};
        [best_gamma_under, corrected_underexposed_image] =
find_best_gamma(original_underexposed_image, gamma_values, target_mean,
target_variance);

        % Visualize original and corrected underexposed image
        figure;
        subplot(1, 2, 1);
        imshow(ycbcr2rgb(original_underexposed_image));
        title('Original Underexposed');

        subplot(1, 2, 2);
        imshow(ycbcr2rgb(corrected_underexposed_image));
```

```
title(['Gamma Corrected (gamma = ', num2str(best_gamma_under),  
'')]);  
end  
end
```

**Original Underexposed**



**Gamma Corrected (gamma = 0.4)**





```
% Process random overexposed images
for i = 1:num_images_to_process
    if ~isempty(overexposed_images)
        original_overexposed_image =
overexposed_images{overexposed_indices(i)};
        [best_gamma_over, corrected_overexposed_image] =
find_best_gamma(original_overexposed_image, gamma_values, target_mean,
target_variance);

        % Visualize original and corrected overexposed image
        figure;
        subplot(1, 2, 1);
        imshow(ycbcr2rgb(original_overexposed_image));
        title('Original Overexposed');

        subplot(1, 2, 2);
        imshow(ycbcr2rgb(corrected_overexposed_image));
        title(['Gamma Corrected (gamma = ', num2str(best_gamma_over), ')']);
    end
end
```

**Original Overexposed**



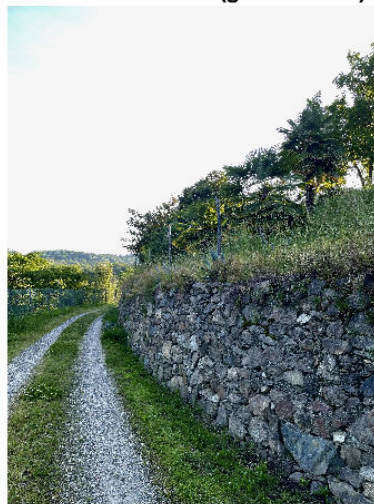
**Gamma Corrected (gamma = 1.8)**



**Original Overexposed**



**Gamma Corrected (gamma = 1.4)**



```
% Process random correctly exposed images
for i = 1:num_images_to_process
```

```

if ~isempty(correctly_exposed_images)
    original_correctly_exposed_image =
correctly_exposed_images{correctly_exposed_indices(i)};
    [best_gamma_correctly_exposed, corrected_correctly_exposed_image]
= find_best_gamma(original_correctly_exposed_image, gamma_values,
target_mean, target_variance);

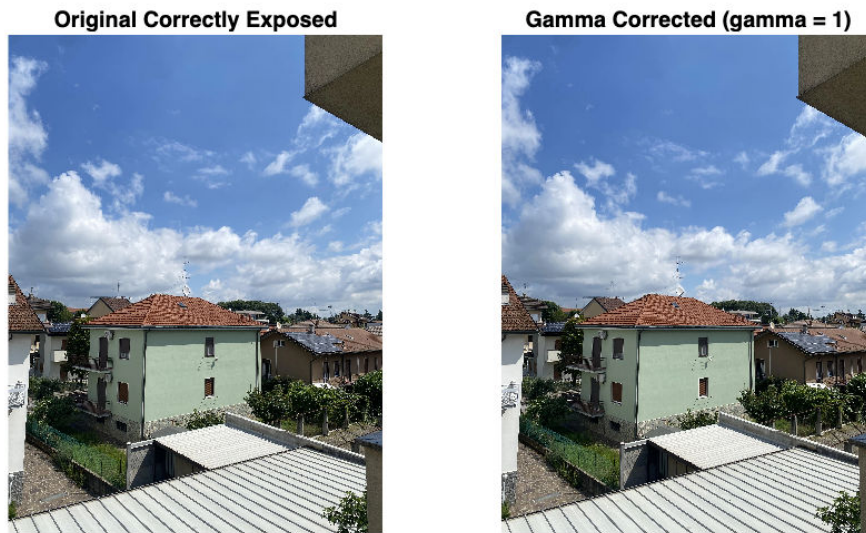
    % Visualize original and corrected correctly exposed image
    figure;
    subplot(1, 2, 1);
    imshow(ycbcr2rgb(original_correctly_exposed_image));
    title('Original Correctly Exposed');

    subplot(1, 2, 2);
    imshow(ycbcr2rgb(corrected_correctly_exposed_image));
    title(['Gamma Corrected (gamma = ',
num2str(best_gamma_correctly_exposed), ')']);
end
end

```







## Section 4: Gamma Correction with Optimization

In this section, we apply gamma correction to underexposed, overexposed, and correctly exposed images using an optimization method based on **RMSE** (Root Mean Square Error) and the **fmincon** function. This method uses **fmincon** to find the optimal gamma value that minimizes the RMSE between the corrected image's luminance statistics and the target mean and variance.

**Setting Gamma Values:** The optimization process starts with an **initial guess** for the gamma value and **adjusts it** to *minimize* the *RMSE*, that we have described in details in the previous section

**Applying Gamma Correction:** The gamma correction is applied to the selected images using the custom function `optimize_gamma`, defined in a separate script. This function uses **fmincon** to optimize the gamma value. The **fmincon** function is a powerful optimization tool in MATLAB that handles constraints and provides robust solutions for nonlinear optimization problems. Compared to **fminsearch**, **fmincon** is preferred because it allows for the specification of **bounds** and **constraints** on the gamma values, ensuring that the solution remains within a *realistic and effective range*.

The logic of the optimization process involves the following steps:

1. **Defining the Objective Function:** The function `evaluate_exposure_gamma` is defined to calculate the RMSE for a given gamma value. It applies the gamma correction to the image using the `gammaCorrectionYCbCr` function, extracts the Y channel, and then evaluates the corrected image using the `evaluate_exposure` function, which computes the *RMSE* based on the differences between the corrected image's mean and variance and the target values.

2. **Setting Initial Conditions and Bounds:** The initial guess for gamma is set to 1.0 (the value for which gamma is not applied, so the most neutral one), with lower and upper bounds ( $lb = 0.1$  and  $ub = 3.0$ ) defined to ensure the gamma value remains within a realistic range (which are the same defined for the brute-force method).
3. **Optimization with fmincon:** The fmincon function is used to minimize the objective function (RMSE). fmincon handles constraints and allows for the specification of bounds, making it a robust choice for this optimization problem. During the optimization process, fmincon iteratively adjusts the gamma value, evaluating the objective function at each step. It calculates the RMSE for the current gamma value and adjusts the gamma to move towards a lower RMSE. This process continues until fmincon converges to a gamma value that minimizes the RMSE. The options for fmincon are set to suppress output (intermediate values/iteration) during the optimization process.
4. **Applying the Optimized Gamma:** Once the optimal gamma value is found, it is applied to the image using the gammaCorrectionYCbCr function to produce the corrected image.

```
% Parameters
num_images_to_process = 2; % Number of random images to process for each
class

% Select random images from each class
if ~isempty(underexposed_images)
    underexposed_indices = randperm(length(underexposed_images),
num_images_to_process);
end

if ~isempty(overexposed_images)
    overexposed_indices = randperm(length(overexposed_images),
num_images_to_process);
end

if ~isempty(correctly_exposed_images)
    correctly_exposed_indices = randperm(length(correctly_exposed_images),
num_images_to_process);
end
```

```
% Process random underexposed images using fmincon
for i = 1:num_images_to_process
    if ~isempty(underexposed_images)
        original_underexposed_image =
underexposed_images{underexposed_indices(i)};
        [best_gamma_under, corrected_underexposed_image] =
optimize_gamma(original_underexposed_image, 0.5, 0.03);

        % Visualize original and corrected underexposed image
        figure;
        subplot(1, 2, 1);
```

```

imshow(ycbcr2rgb(original_underexposed_image));
title('Original Underexposed');

subplot(1, 2, 2);
imshow(ycbcr2rgb(corrected_underexposed_image));
title(['Optimized Gamma (gamma = ', num2str(best_gamma_under),
')']);
end
end

```

**Original Underexposed**



**Optimized Gamma (gamma = 0.40574)**





Original Underexposed



Optimized Gamma (gamma = 0.38407)



```
% Process random overexposed images using fmincon
for i = 1:num_images_to_process
    if ~isempty(overexposed_images)
        original_overexposed_image =
overexposed_images{overexposed_indices(i)};
        [best_gamma_over, corrected_overexposed_image] =
optimize_gamma(original_overexposed_image, 0.5, 0.03);

        % Visualize original and corrected overexposed image
        figure;
        subplot(1, 2, 1);
        imshow(ycbcr2rgb(original_overexposed_image));
        title('Original Overexposed');

        subplot(1, 2, 2);
        imshow(ycbcr2rgb(corrected_overexposed_image));
        title(['Optimized Gamma (gamma = ', num2str(best_gamma_over), ')']);
    end
end
```

Original Overexposed



Optimized Gamma (gamma = 2.4163)



Original Overexposed



Optimized Gamma (gamma = 2.0965)



```
% Process random correctly exposed images using fmincon  
for i = 1:num_images_to_process
```

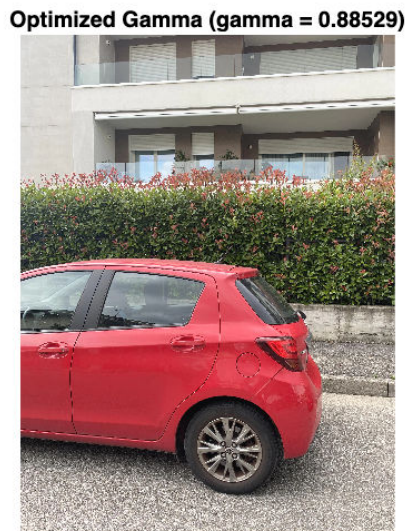
```

if ~isempty(correctly_exposed_images)
    original_correctly_exposed_image =
correctly_exposed_images{correctly_exposed_indices(i)};
    [best_gamma_correctly_exposed, corrected_correctly_exposed_image] =
optimize_gamma(original_correctly_exposed_image, 0.5, 0.03);

    % Visualize original and corrected correctly exposed image
    figure;
    subplot(1, 2, 1);
    imshow(ycbcr2rgb(original_correctly_exposed_image));
    title('Original Correctly Exposed');

    subplot(1, 2, 2);
    imshow(ycbcr2rgb(corrected_correctly_exposed_image));
    title(['Optimized Gamma (gamma = ',
num2str(best_gamma_correctly_exposed), ')']);
end
end

```





## Section 5: Process and Save Images

In this section, we **process** all images using the optimization method based on `fmincon` and RMSE, and **save** the gamma-corrected images in a specified output folder. The process involves ensuring the *existence of the output folder*, processing each set of images (underexposed, overexposed, and correctly exposed) using the `save_corrected_images` function, and saving the corrected images.

The **output folder** is specified using the `fullfile` function to create a path in the current working directory. If the folder does not exist, it is created using the `mkdir` function, and a confirmation message is printed. If the folder already exists, the script proceeds without creating a new one.

The function `save_corrected_images` processes each image using the `optimize_gamma` function to **find** the **best gamma** value and **save** the **corrected image** in the output folder. It first **checks** if the output folder exists, and if not, it prints a message and exits. For each image, `optimize_gamma` is called to determine the optimal gamma value, which is then applied to the image. The corrected image is converted back to RGB format, and a new filename is generated with the prefix 'corrected\_'. The corrected image is then saved in the specified output folder.

```
% Parameters
output_folder = fullfile(pwd, 'corrected_images'); % Specify your output
folder, here it is created in the current working directory

% Ensure the output folder exists or create it
if ~exist(output_folder, 'dir')
    mkdir(output_folder);
```

```

    fprintf('Created output folder: %s\n', output_folder);
end

% Process and save selected underexposed images using fmincon
if ~isempty(underexposed_images)
    save_corrected_images(underexposed_images, underexposed_filenames,
1:length(underexposed_images), output_folder, 0.47, 0.039);
end

```

```

% Process and save selected overexposed images using fmincon
if ~isempty(overexposed_images)
    save_corrected_images(overexposed_images, overexposed_filenames,
1:length(overexposed_images), output_folder, 0.47, 0.039);
end

```

```

% Process and save selected correctly exposed images using fmincon
if ~isempty(correctly_exposed_images)
    save_corrected_images(correctly_exposed_images,
correctly_exposed_filenames, 1:length(correctly_exposed_images),
output_folder, 0.47, 0.039);
end

```

## Section 6: Upload and Classify the Corrected Images

We repeat what we have already done for the classification part. We extract the 10 most significant features through the function `extract_feature_gamma` from the corrected images. We upload the best classifier `trainedSVM` and we predict the new results.

```

% Define the path from th local machine
path_gamma = './corrected_images';
% Define the directory containing our dataset
dataset_gamma = dir(fullfile(path_gamma, '*.jpg'));

% Define the names
names_gamma = {dataset_gamma.name}';

% Create an array of paths
paths_gamma = cellfun(@(s) fullfile(path_gamma, s), names_gamma,
'UniformOutput', false);
disp(numel(names_gamma))

```

138

```

% Extract the features in corrected images with gamma
features_gamma = extract_feat(paths_gamma, @extract_features_gamma);

```

```

20/138 (14.49%)
40/138 (28.99%)
60/138 (43.48%)

```

```
80/138 (57.97%)
100/138 (72.46%)
120/138 (86.96%)
```

```
disp(size(features_gamma))
```

```
138    10
```

```
% Example of the first 2 rows
```

```
disp(features_gamma(1:2,:));
```

```
    0.4929    0.7351    0.8756    0.5496    0.0500    0.6377    0.3187    0.3913    0.3256    0.2627
    0.6030    0.3812    0.7656    0.6639    0.0819    0.4591    0.4344    0.1527    0.0167    0.3414
```

```
% Load the trained model from the .mat file if not already loaded
```

```
load('trainedSVM.mat', 'trainedSVM', 'validationAccuracy');
```

```
% Predict on new data using the loaded model
```

```
predictions_gamma = trainedSVM.predictFcn(features_gamma)
```

```
% Assuming saved_corrected_images is a cell array or an array with images
```

```
num_instances = numel(predictions_gamma);
```

We calculate the **accuracy**, **precision**, **recall**, **f1** and **confusion matrix** of the new predictions.

```
% Create the new targets vector of all ones
```

```
targets_gamma = ones(num_instances, 1);
```

```
% Calculate the accuracy of the new predictions
```

```
accuracy_gamma = sum(predictions_gamma == targets_gamma) /  
numel(targets_gamma);
```

```
% Display the accuracy
```

```
disp(['New Data Accuracy: ', num2str(accuracy_gamma * 100), '%']);
```

```
New Data Accuracy: 82.6087%
```

```
% Confusion matrix
```

```
conf_matrix = confusionmat(targets_gamma, predictions_gamma);
```

```
figure, confusionchart(conf_matrix)
```



1	114	17	7
2			
3			
	1	2	3

Predicted Class

```
% Precision, Recall, F1 Score
precision = diag(conf_matrix) ./ sum(conf_matrix, 2);
recall = diag(conf_matrix) ./ sum(conf_matrix, 1)';
f1 = 2 * (precision .* recall) ./ (precision + recall);

disp('Precision for each class:'), disp(precision');
```

```
Precision for each class:
    0.8261      NaN      NaN
```

```
disp('Recall for each class:'), disp(recall');
```

```
Recall for each class:
     1     0     0
```

```
disp('F1 Score for each class:'), disp(f1');
```

```
F1 Score for each class:
    0.9048      NaN      NaN
```

## Section Bonus

### Entropy as and Evaluation Method of Gamma Corrected Images

#### Mean and Variance of Entropy from Correctly Exposed Images

In this section, we calculate the **entropy** of the luminance channel (Y) from correctly exposed images.

Entropy measures the amount of **information** or **randomness** present in the image. *Higher* entropy indicates more *details* and *texture*, while *lower* entropy indicates a *smoother*, more *uniform* image. So it quantifies the amount of detail and contrast in the image. By comparing the entropy of gamma-corrected images to the target entropy derived from correctly exposed images, we can assess whether the gamma correction has appropriately enhanced the image's details without introducing excessive noise or losing important information.

For each correctly exposed image, the **luminance** channel (Y) is extracted, and the entropy values are computed using a custom function `calculate_entropy`. These values are then stored in the array `norm_entropy`.

Next, the mean and variance of the entropy values are calculated to obtain representative metrics for the entire dataset of correctly exposed images. These metrics, `target_mean_entropy` and `target_variance_entropy`, are printed for verification. The acceptable range of entropy values is defined as one standard deviation around the mean, providing a benchmark for evaluating the entropy of gamma-corrected images.

To visualize the distribution of the target entropy, a normal distribution is approximated using the calculated mean and variance.

```
% Calculate the entropy of the correctly exposed original images
norm_entropy = zeros(1, num_correctly_to_process);

for i = 1:num_correctly_to_process
    img_y = correctly_exposed_images{i}(:,:,1);
    norm_entropy(i) = calculate_entropy(img_y);
end

% Calculate the mean and variance of the entropy
target_mean_entropy = mean(norm_entropy);
target_variance_entropy = var(norm_entropy);

% Define the acceptable entropy range (two standard deviations)
entropy_range = [target_mean_entropy - 2 * sqrt(target_variance_entropy),
    ...
                 target_mean_entropy + 2 * sqrt(target_variance_entropy)];

% Print the final target mean
fprintf('Target Mean: %f\n', target_mean_entropy)
```

Target Mean: 7.343380

```
% Print the final target variance
fprintf('Target Variance: %f\n', target_variance_entropy)
```

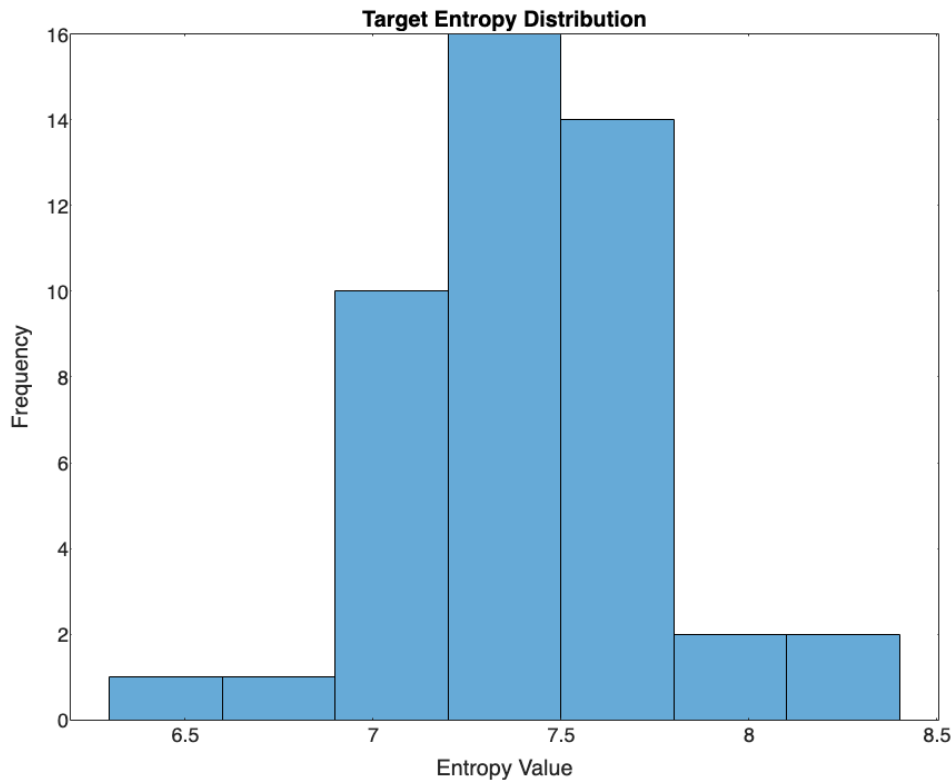
Target Variance: 0.065137

```
% Fix the seed for the random number generator for reproducibility
rng(0);

% Visualize target entropy distribution
```



```
target_dist_entropy = makedist("Normal", "mu", target_mean_entropy,
"sigma", sqrt(target_variance_entropy));
target_entropy_values = random(target_dist_entropy, size(norm_entropy));
figure, histogram(target_entropy_values), title('Target Entropy
Distribution'), xlabel('Entropy Value'), ylabel('Frequency');
```



## Evaluating Entropy of Gamma-Corrected Images

In this section, we calculate the entropy of the luminance channel (Y) from gamma-corrected images. By comparing the entropy of gamma-corrected images to the target entropy, we can assess whether the gamma correction has appropriately enhanced the image's details without introducing excessive noise or losing important information.

- First the corrected images are **read** from the specified directory (path\_gamma).
- Then the **filenames** and **full paths** of the corrected images are obtained using `dir` and `fullfile` functions. Arrays are initialized to store the entropy values of the corrected images and to count the number of acceptable and unacceptable images.
- For each corrected image, the image is read and converted to the **YCbCr** color space. The Y (luminance) channel is extracted, and its **entropy** is calculated using the `calculate_entropy` function.
- The calculated entropy value for each image is **compared** to the predefined acceptable range.
- The number of acceptable and unacceptable images is counted based on whether the entropy value falls within this range. The **results** are displayed, showing the number of acceptable and unacceptable images and the entropy value of the last corrected image.

```

% Calculate the entropy of the corrected images
corrected_files = dir(fullfile(path_gamma, '*.jpg'));
corrected_filenames = {corrected_files.name};
corrected_paths = fullfile(path_gamma, corrected_filenames);

corrected_entropy = zeros(1, length(corrected_paths));
num_acceptable = 0;
num_unacceptable = 0;

for i = 1:length(corrected_paths)
    corrected_img = imread(corrected_paths{i});
    corrected_img_ycbcr = rgb2ycbcr(corrected_img);
    corrected_img_y = corrected_img_ycbcr(:,:,1);
    corrected_entropy(i) = calculate_entropy(corrected_img_y);

    if corrected_entropy(i) >= entropy_range(1) && corrected_entropy(i) <=
entropy_range(2)
        num_acceptable = num_acceptable + 1;
    else
        num_unacceptable = num_unacceptable + 1;
    end
end

% Display the results
fprintf('Number of acceptable images: %d\n', num_acceptable);

```

Number of acceptable images: 93

```

fprintf('Number of unacceptable images: %d\n', num_unacceptable);

```

Number of unacceptable images: 45

```

% Display the entropy value of the last image
fprintf('Entropy of the last corrected image: %.2f\n',
corrected_entropy(end));

```

Entropy of the last corrected image: 7.67