OKANAGAN

ENGR 418 Final Project

Lego Image Classification

Tyson Senger (29355971)

School of Engineering Faculty of Applied Science
The University of British Columbia Okanagan

Submission date: December 13, 2023

1.0 Introduction

In response to the evolving needs of the client, I am tasked with enhancing the existing sorting algorithm. I am provided with a new training and testing set of images, which this time are not centered or oriented in a given direction. These pictures present a more realistic view of what the sorting algorithm will encounter in the factory. My algorithm must evolve to classify the four different classes despite their centering and orientations. To achieve this, I will extract and engineer features from the images. I will create this algorithm in accordance with the client's specifications: a four-class Logistic Regression classifier using a maximum of 64 features per image. My objective is to deliver an algorithm that will outperform its predecessor when the centering and orientation become variable.

2.0 Theory

Since I must use a multi-class Logistic Regression classifier for this algorithm, I need to understand how it works. The model takes inputs, which in this case will be engineered characteristics extracted from RGB images of Lego pieces. Then, the model associates weights and biases to each feature, using the dot product of the feature and weight vector, plus the bias term, to create a linear combination of the terms. These linear combinations are passed through the softmax function, which normalizes the scores into probabilities. The class with the highest probability is then predicted as the output class for that input. In my case, this will involve assigning one of the four shape classes to an input Lego image. During training, the model's weights and biases will be optimized using a loss function to minimize the difference between the model's predicted probabilities and the class labels given along with each image in the training data.

3.0 Algorithm

3.1 Pseudocode

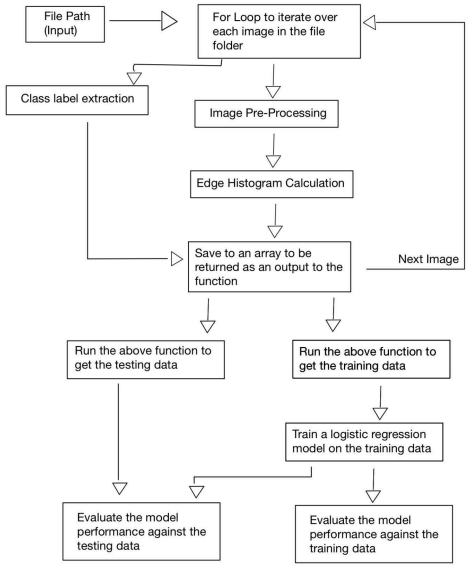

Figure 3.1.1

3.2 Data Pre-Processing and Feature Engineering

Each image is initially loaded into a variable based on its file path. Each image is scaled to 1000 x 1000 pixels for uniformity. The image is then converted to the HSV color space (hue, saturation, brightness value). This is because the saturation channel of the HSV color space behaves similarly to a grayscale image but also provides a very clean edge for the edge detection function to use. I then boost the contrast using the ImageEnhance.Contrast() library.

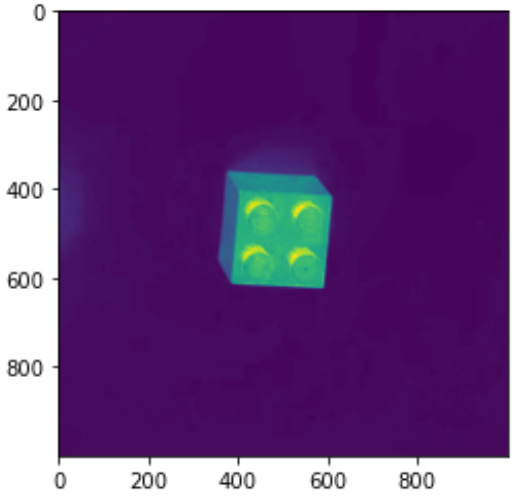

Figure 3.1.2: HSV Colour Space Saturation Channel

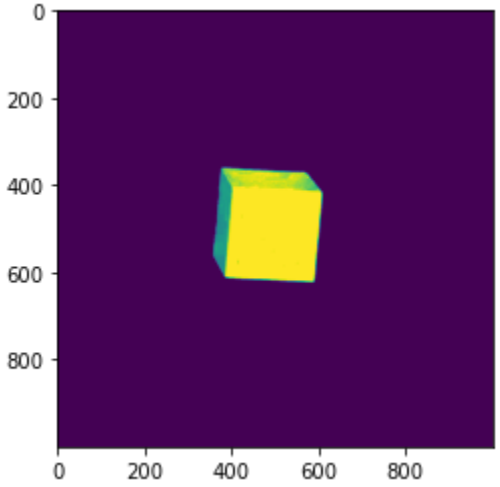

Figure 3.1.3: Saturation Channel, Contrast Boosted

Using the pixel data illustrated in *Figure 3.1.3*, I apply the ImageFilter.FIND_EDGES function, which outputs an image consisting of the edges of the object in the image. I then take that pixel data and apply a threshold calculation that sets all pixel values lower than 100 to 0. This allows me to crop each image based on where the object is in the frame, with a margin of 50 pixels on all sides of the object.

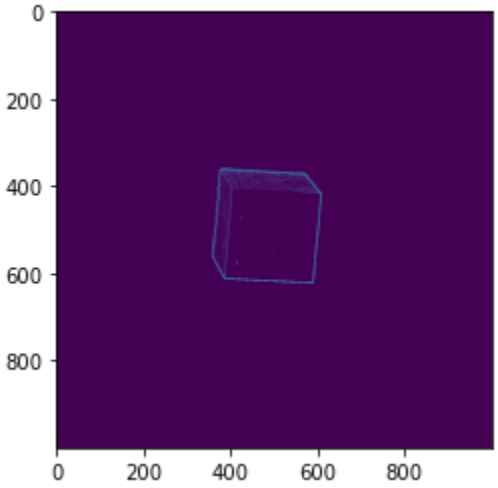

Figure 3.1.4: Edge Detection Image

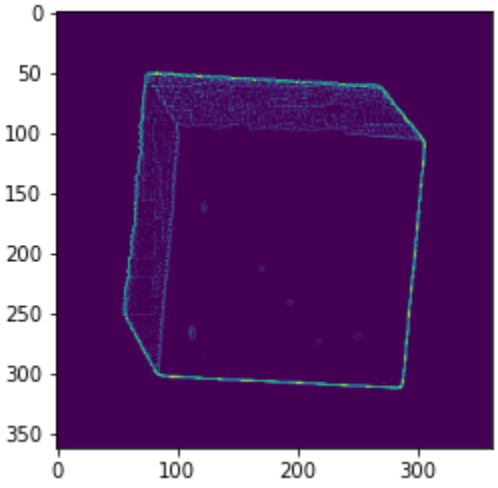

Figure 3.1.5: Custom Cropping With Margin = 50 Pixels

Next, I sum each column of the rotated arrays to find the pixel frequency for that column. I do this four times, rotating the image 30 degrees each time. This way, I detect lines at 0 degrees, 30 degrees, 60 degrees, and 90 degrees. I concatenate these findings for each image, so I now have a histogram for each image that consists of the edge frequency of the pixels at various angles. However, I am allotted a maximum of 64 features per image. Since I am detecting lines at 4 different angles, I can have a maximum of 16 frequency bins per rotation per image. Because of this, I created the rebin_histogram() function, which takes the original histogram and reduces the number of frequency bins while maintaining as much frequency data as possible. As seen in *Figure 3.1.7*, the number of frequency bins is 64, and the 4 distinct bin categories correspond to the four rotations of the edge detection algorithm.

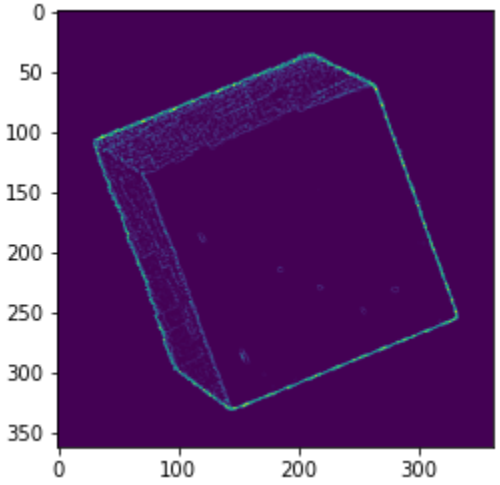

Figure 3.1.6: Edge Image Rotated 30°

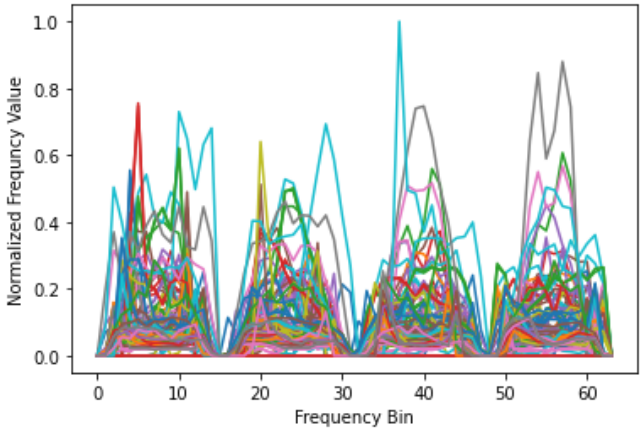

Figure 3.1.7: Edge Histogram (Re-binned) (Each Line Corresponds to a Single Image)

Next, I load the class label for each image into an array that will be saved as the class variable to be used by the logistic regression model. This is done by loading the file name portion of the file path and decoding the class value from the file name.

3.4 Model Parameters

Using the GridSearchCV() library, I iterated through several parameters for the Logistic Regression model. This part of the code can take several minutes to run and was therefore removed from the code submission. However, it found that for my purposes the optimal parameters are as follows in *Figure 3.2.1*:

Logistic Regressor:	Multi-class	Solver	C	Penalty	Class Weight	Max Iteration
Optimal Parameters	Multinomial	lbfgs	100	L2	Balanced	1000

Figure 3.2.1

4.0 Results and Discussion

4.1 Accuracies and Confusion Matrices

Stage 1	Accuracy	100%		
Training	Actual Class			
Predicted Class	2x1	Cir	Rec	Squ
2x1	27	0	0	0
Cir	0	27	0	0
Rec	0	0	27	0
Squ	0	0	0	27

Figure 4.1: Stage 1 Code Performance Training Data

Stage 1	Accuracy	67.50%		
Testing	Actual Class			
Predicted Class	2x1	Cir	Rec	Squ
2x1	21	2	4	0
Cir	5	16	5	1
Rec	0	9	17	1
Squ	2	2	4	19

Figure 4.2: Stage 1 Code Performance Testing Data

Stage 2	Accuracy	92.60%		
Training	Actual Class			
Predicted Class	2x1	Cir	Rec	Squ
2x1	25	2	0	0
Cir	1	26	0	0
Rec	0	0	27	0
Squ	0	5	0	22

Figure 4.3: Stage 2 Code Performance Training Data

Stage 2	Accuracy	69.40%		
Testing	Actual Class			
Predicted Class	2x1	Cir	Rec	Squ
2x1	8	7	6	6
Cir	2	19	2	4
Rec	1	0	26	0
Squ	1	4	0	22

Figure 4.4: Stage 2 Code Performance Testing Data

4.2 Model Comparison

As can be seen above, the stage two code slightly outperforms the stage one code. It is important to note, however, that the stage one code was permitted 4096 features per image, while the stage two code was allowed 64. Taking this into account, the stage two code vastly outperformed the stage one code.

4.3 Error Analysis

The class of Lego that my model was most likely to incorrectly classify is the ‘2x1’ piece, with 8 true positives. It mistook the ‘2x1’ piece for the ‘Cir’ piece 7 times and the ‘Rec’ and ‘Squ’ pieces 6 times each. This makes sense when considering that when I cropped the image to focus on the subject of the image, I kept the number of margin pixels the same for each image instead of retaining the cropped image pixel count. This means that when comparing the 2x1 and Rec pieces, even though they are different size pieces since they are both ‘rectangular’, they will appear very similar after being cropped.

The class that my model most accurately predicted was the Rec Lego piece. It is hard to say for certain why the model performed so well on the Rec pieces, but one reason may be that Rec images in the training and testing folder, on average, were better lit, more centered, and more aligned than the rest of the classes. I can't prove this, but looking at *Figure 4.1*, I think it is possible.

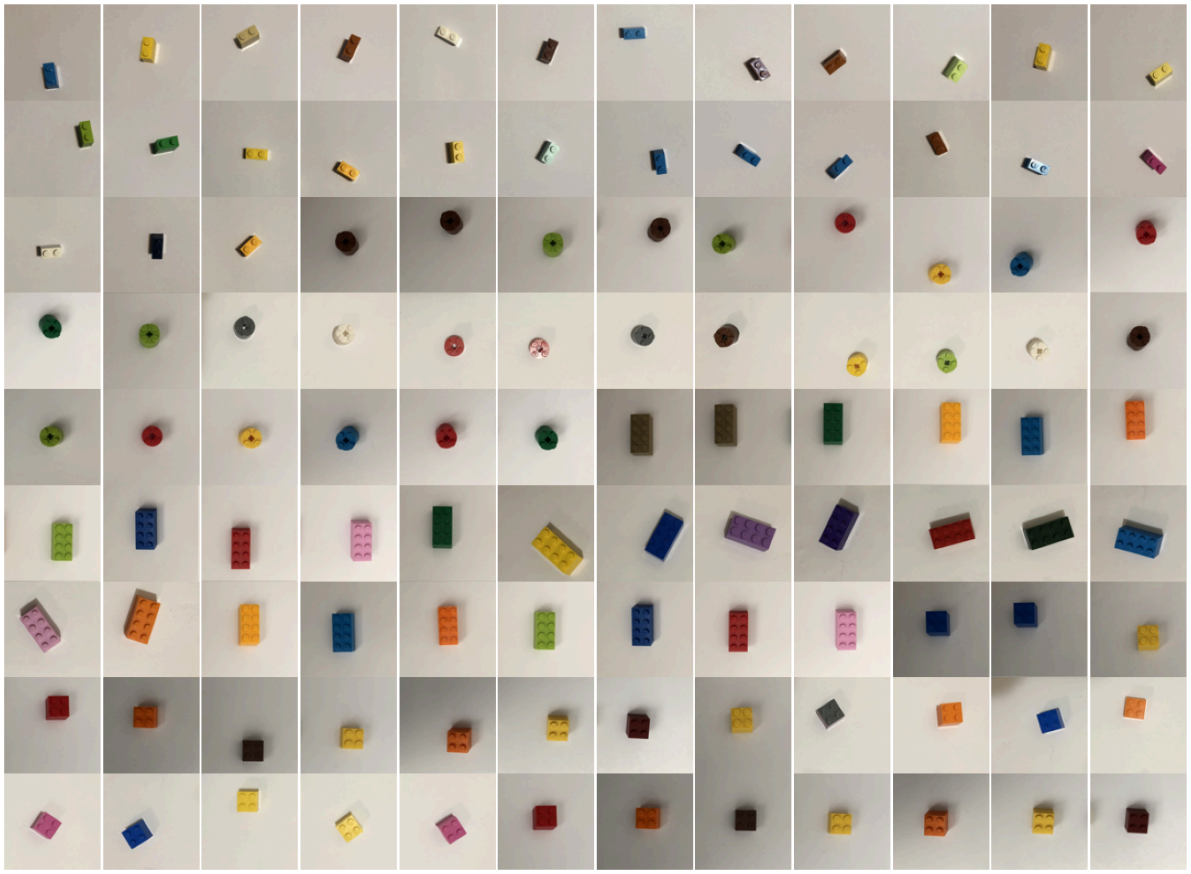

Figure 4.1: All Training Images

5.0 Conclusion

During the coding process, I noticed that the order in which you execute each image pre-processing calculation is important. To retain the most amount of image information, I decided to first convert color spaces, then increase contrast, then eliminate the low-value pixel noise. After that stage of pre-processing, I implemented the edge detection library. Then, after the first round of edge detection, I cropped and rotated the image for the second round of edge detection. However, the cropping of the image may have inadvertently led to more misclassifications, as the different sized pieces would look similar after cropping. Since the maximum number of features was 64, these steps were deemed necessary to reduce the number of features efficiently. An algorithm included in my code to re-center and reorient the Lego piece like the initial set of training and testing images could have potentially led to higher accuracy, but the reduced feature size would still need to be considered.