Traffic_Light_Classifier

October 17, 2018

0.1 # Traffic Light Classifier

In this project, you'll use your knowledge of computer vision techniques to build a classifier for images of traffic lights! You'll be given a dataset of traffic light images in which one of three lights is illuminated: red, yellow, or green.

In this notebook, you'll pre-process these images, extract features that will help us distinguish the different types of images, and use those features to classify the traffic light images into three classes: red, yellow, or green. The tasks will be broken down into a few sections:

1. **Loading and visualizing the data**. The first step in any classification task is to be familiar with your data; you'll need to load in the images of traffic lights and visualize them!

2. **Pre-processing**. The input images and output labels need to be standardized. This way, you can analyze all the input images using the same classification pipeline, and you know what output to expect when you eventually classify a *new* image.

3. **Feature extraction**. Next, you'll extract some features from each image that will help distinguish and eventually classify these images.

4. **Classification and visualizing error**. Finally, you'll write one function that uses your features to classify *any* traffic light image. This function will take in an image and output a label. You'll also be given code to determine the accuracy of your classification model.

5. **Evaluate your model**. To pass this project, your classifier must be >90% accurate and never classify any red lights as green; it's likely that you'll need to improve the accuracy of your classifier by changing existing features or adding new features. I'd also encourage you to try to get as close to 100% accuracy as possible!

Here are some sample images from the dataset (from left to right: red, green, and yellow traffic lights):

0.1.1 *Here's what you need to know to complete the project:*

Some template code has already been provided for you, but you'll need to implement additional code steps to successfully complete this project. Any code that is required to pass this project is marked with **'(IMPLEMENTATION)'** in the header. There are also a couple of questions about your thoughts as you work through this project, which are marked with **'(QUESTION)'** in the

header. Make sure to answer all questions and to check your work against the project rubric to make sure you complete the necessary classification steps!

Your project submission will be evaluated based on the code implementations you provide, and on two main classification criteria. Your complete traffic light classifier should have: 1. **Greater than 90% accuracy** 2. ***Never*** **classify red lights as green**

1 1. Loading and Visualizing the Traffic Light Dataset

This traffic light dataset consists of 1484 number of color images in 3 categories - red, yellow, and green. As with most human-sourced data, the data is not evenly distributed among the types. There are: * 904 red traffic light images * 536 green traffic light images * 44 yellow traffic light images

Note: All images come from this MIT self-driving car course and are licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

1.0.1 Import resources

Before you get started on the project code, import the libraries and resources that you'll need.

```
In [1]: # Zip all files to Download Notebook files
        #!tar cvfz allfiles.tar.gz *
```

```
In [2]: import cv2 # computer vision library
        import helpers # helper functions

        import random
        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.image as mpimg # for loading in images

        %matplotlib inline
```

1.1 Training and Testing Data

All 1484 of the traffic light images are separated into training and testing datasets.

- 80% of these images are training images, for you to use as you create a classifier.
- 20% are test images, which will be used to test the accuracy of your classifier.
- All images are pictures of 3-light traffic lights with one light illuminated.

1.2 Define the image directories

First, we set some variables to keep track of some where our images are stored:

```
IMAGE_DIR_TRAINING: the directory where our training image data is stored
IMAGE_DIR_TEST: the directory where our test image data is stored
```

```
In [3]: # Image data directories
        IMAGE_DIR_TRAINING = "traffic_light_images/training/"
        IMAGE_DIR_TEST = "traffic_light_images/test/"
```

1.3 Load the datasets

These first few lines of code will load the training traffic light images and store all of them in a variable, `IMAGE_LIST`. This list contains the images and their associated label ("red", "yellow", "green").

You are encouraged to take a look at the `load_dataset` function in the helpers.py file. This will give you a good idea about how lots of image files can be read in from a directory using the glob library. The `load_dataset` function takes in the name of an image directory and returns a list of images and their associated labels.

For example, the first image-label pair in `IMAGE_LIST` can be accessed by index: `IMAGE_LIST[0][:]`.

```
In [4]: # Using the load_dataset function in helpers.py
        # Load training data
        IMAGE_LIST = helpers.load_dataset(IMAGE_DIR_TRAINING)
```

1.4 Visualize the Data

The first steps in analyzing any dataset are to 1. load the data and 2. look at the data. Seeing what it looks like will give you an idea of what to look for in the images, what kind of noise or inconsistencies you have to deal with, and so on. This will help you understand the image dataset, and **understanding a dataset is part of making predictions about the data**.

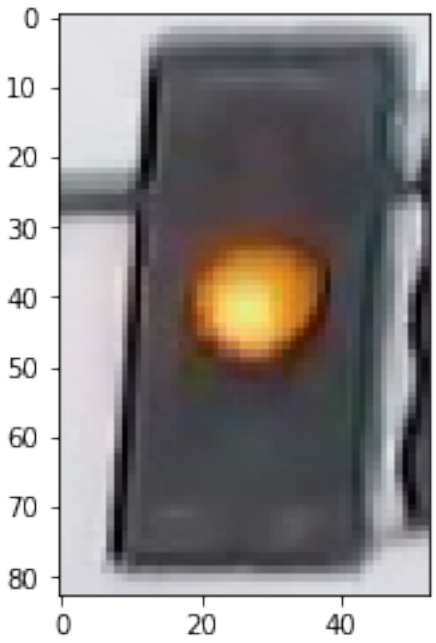
1.4.1 Visualize the input images

Visualize and explore the image data! Write code to display an image in `IMAGE_LIST`: * Display the image * Print out the shape of the image * Print out its corresponding label

See if you can display at least one of each type of traffic light image – red, green, and yellow — and look at their similarities and differences.

```
In [5]: ## TODO: Write code to display an image in IMAGE_LIST (try finding a yellow traffic ligh
        ## TODO: Print out 1. The shape of the image and 2. The image's label
        selected_image = IMAGE_LIST[730][0]
        plt.imshow(selected_image)
        print("Shape: " + str(selected_image.shape))
        print("Label: " + IMAGE_LIST[730][1])

        # The first image in IMAGE_LIST is displayed below (without information about shape or l
        #selected_image = IMAGE_LIST[0][0]
        #plt.imshow(selected_image)
```

```
Shape: (83, 53, 3)
Label: yellow
```



2 2. Pre-process the Data

After loading in each image, you have to standardize the input and output!

2.0.1 Input

This means that every input image should be in the same format, of the same size, and so on. We'll be creating features by performing the same analysis on every picture, and for a classification task like this, it's important that **similar images create similar features**!

2.0.2 Output

We also need the output to be a label that is easy to read and easy to compare with other labels. It is good practice to convert categorical data like "red" and "green" to numerical data.

A very common classification output is a 1D list that is the length of the number of classes - three in the case of red, yellow, and green lights - with the values 0 or 1 indicating which class a certain image is. For example, since we have three classes (red, yellow, and green), we can make a list with the order: [red value, yellow value, green value]. In general, order does not matter, we choose the order [red value, yellow value, green value] in this case to reflect the position of each light in descending vertical order.

A red light should have the label: [1, 0, 0]. Yellow should be: [0, 1, 0]. Green should be: [0, 0, 1]. These labels are called **one-hot encoded labels**.

(Note: one-hot encoding will be especially important when you work with machine learning algorithms).

(IMPLEMENTATION): Standardize the input images

- Resize each image to the desired input size: 32x32px.
- (Optional) You may choose to crop, shift, or rotate the images in this step as well.

It's very common to have square input sizes that can be rotated (and remain the same size), and analyzed in smaller, square patches. It's also important to make all your images the same size so that they can be sent through the same pipeline of classification steps!

```
In [6]: # This function should take in an RGB image and return a new, standardized version
        def standardize_input(image):

            ## TODO: Resize image and pre-process so that all "standard" images are the same siz
            standard_im = np.copy(image)
            standard_im = cv2.resize(standard_im, (32, 32))
            return standard_im
```

2.1 Standardize the output

With each loaded image, we also specify the expected output. For this, we use **one-hot encoding**.

- One-hot encode the labels. To do this, create an array of zeros representing each class of traffic light (red, yellow, green), and set the index of the expected class number to 1.

Since we have three classes (red, yellow, and green), we have imposed an order of: [red value, yellow value, green value]. To one-hot encode, say, a yellow light, we would first initialize an array to [0, 0, 0] and change the middle value (the yellow value) to 1: [0, 1, 0].

(IMPLEMENTATION): Implement one-hot encoding

```
In [7]: ## TODO: One hot encode an image label
        ## Given a label - "red", "green", or "yellow" - return a one-hot encoded label

        # Examples:
        # one_hot_encode("red") should return: [1, 0, 0]
        # one_hot_encode("yellow") should return: [0, 1, 0]
        # one_hot_encode("green") should return: [0, 0, 1]

        def one_hot_encode(label):
            ## TODO: Create a one-hot encoded label that works for all classes of traffic lights
            #if label == "red":
            #    return [1,0,0]
            #elif label == "yellow":
            #    return [0,1,0]
            #else:
            #    return [0,0,1]
```

```
    label_types = ['red', 'yellow', 'green']
    # Create a vector of 0's that is the length of the number of classes (3)
    one_hot_encoded = [0] * len(label_types)

    # Set the index of the class number to 1
    one_hot_encoded[label_types.index(label)] = 1

    return one_hot_encoded
```

2.1.1 Testing as you Code

After programming a function like this, it's a good idea to test it, and see if it produces the expected output. **In general, it's good practice to test code in small, functional pieces, after you write it**. This way, you can make sure that your code is correct as you continue to build a classifier, and you can identify any errors early on so that they don't compound.

All test code can be found in the file `test_functions.py`. You are encouraged to look through that code and add your own testing code if you find it useful!

One test function you'll find is: `test_one_hot(self, one_hot_function)` which takes in one argument, a one_hot_encode function, and tests its functionality. If your one_hot_label code does not work as expected, this test will print ot an error message that will tell you a bit about why your code failed. Once your code works, this should print out TEST PASSED.

```
In [8]: # Importing the tests
        import test_functions
        tests = test_functions.Tests()

        # Test for one_hot_encode function
        tests.test_one_hot(one_hot_encode)
```

TEST PASSED

2.2 Construct a `STANDARDIZED_LIST` of input images and output labels.

This function takes in a list of image-label pairs and outputs a **standardized** list of resized images and one-hot encoded labels.

This uses the functions you defined above to standardize the input and output, so those functions must be complete for this standardization to work!

```
In [9]: def standardize(image_list):

            # Empty image data array
            standard_list = []

            # Iterate through all the image-label pairs
            for item in image_list:
                image = item[0]
                label = item[1]
```

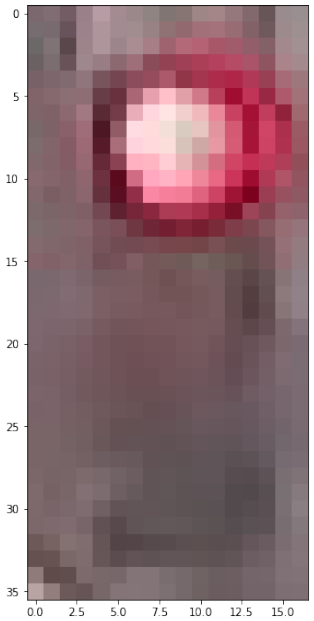
```
            # Standardize the image
            standardized_im = standardize_input(image)

            # One-hot encode the label
            one_hot_label = one_hot_encode(label)

            # Append the image, and it's one hot encoded label to the full, processed list o
            standard_list.append((standardized_im, one_hot_label))

        return standard_list

    # Standardize all training images
    STANDARDIZED_LIST = standardize(IMAGE_LIST)
```

2.3 Visualize the standardized data

Display a standardized image from STANDARDIZED_LIST and compare it with a non-standardized image from IMAGE_LIST. Note that their sizes and appearance are different!

In [10]:
```
## TODO: Display a standardized image and its label
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))

num=0;
selected_image = STANDARDIZED_LIST[num][0]
ax1.imshow(selected_image)
print (selected_image.shape)
print (STANDARDIZED_LIST[num][1])

num=0;
selected_image = IMAGE_LIST[num][0]
ax2.imshow(selected_image)
print (selected_image.shape)
print (IMAGE_LIST[num][1])
```

```
(32, 32, 3)
[1, 0, 0]
(36, 17, 3)
red
```



3 3. Feature Extraction

You'll be using what you now about color spaces, shape analysis, and feature construction to create features that help distinguish and classify the three types of traffic light images.

You'll be tasked with creating **one feature** at a minimum (with the option to create more). The required feature is **a brightness feature using HSV color space**:

1. A brightness feature.
    - Using HSV color space, create a feature that helps you identify the 3 different classes of traffic light.
    - You'll be asked some questions about what methods you tried to locate this traffic light, so, as you progress through this notebook, always be thinking about your approach: what works and what doesn't?
2. (Optional): Create more features!

Any more features that you create are up to you and should improve the accuracy of your traffic light classification algorithm! One thing to note is that, to pass this project you must **never classify a red light as a green light** because this creates a serious safety risk for a self-driving car. To avoid this misclassification, you might consider adding another feature that specifically distinguishes between red and green lights.

These features will be combined near the end of his notebook to form a complete classification algorithm.

3.1 Creating a brightness feature

There are a number of ways to create a brightness feature that will help you characterize images of traffic lights, and it will be up to you to decide on the best procedure to complete this step. You should visualize and test your code as you go.

Pictured below is a sample pipeline for creating a brightness feature (from left to right: standardized image, HSV color-masked image, cropped image, brightness feature):

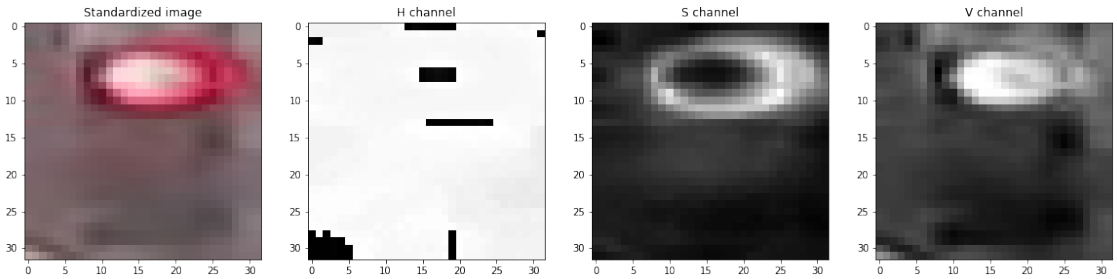
3.2 RGB to HSV conversion

Below, a test image is converted from RGB to HSV colorspace and each component is displayed in an image.

In [11]:
```
# Convert and image to HSV colorspace
# Visualize the individual color channels

image_num = 0
test_im = STANDARDIZED_LIST[image_num][0]
test_label = STANDARDIZED_LIST[image_num][1]

# Convert to HSV
hsv = cv2.cvtColor(test_im, cv2.COLOR_RGB2HSV)

# Print image label
print('Label [red, yellow, green]: ' + str(test_label))

# HSV channels
h = hsv[:,:,0]
s = hsv[:,:,1]
v = hsv[:,:,2]

# Plot the original image and the three channels
f, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(20,10))
ax1.set_title('Standardized image')
ax1.imshow(test_im)
ax2.set_title('H channel')
ax2.imshow(h, cmap='gray')
ax3.set_title('S channel')
ax3.imshow(s, cmap='gray')
ax4.set_title('V channel')
ax4.imshow(v, cmap='gray')
```

```
Label [red, yellow, green]: [1, 0, 0]
```

Out[11]:
```
<matplotlib.image.AxesImage at 0x7f75bbc499e8>
```



(IMPLEMENTATION): Create a brightness feature that uses HSV color space

Write a function that takes in an RGB image and returns a 1D feature vector and/or single value that will help classify an image of a traffic light. The only requirement is that this function should apply an HSV colorspace transformation, the rest is up to you.

From this feature, you should be able to estimate an image's label and classify it as either a red, green, or yellow traffic light. You may also define helper functions if they simplify your code.

```
In [33]: ## TODO: Create a brightness feature that takes in an RGB image and outputs a feature v
         ## This feature should use HSV colorspace values

         def create_feature(rgb_image):
             hsv = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2HSV)
             sum_saturation = np.sum(hsv[:,:,1]) # Sum the saturation values
             area = 32*32
             avg_saturation = sum_saturation / area # Find the average
             return avg_saturation
```

3.3 (Optional) Create more features to help accurately label the traffic light images

```
In [13]: # (Optional) Add more image analysis and create more features
```

3.4 (QUESTION 1): How do the features you made help you distinguish between the 3 classes of traffic light images?

Answer: Seeing some images you could infer that the most distinguish elements over all images is the intense light (red or green or yellow). The background and the traffic light frame is black or it is a very close color to it. So the easiest approach would be to detect based the color variance over Value channel in HSV image type.

4 4. Classification and Visualizing Error

Using all of your features, write a function that takes in an RGB image and, using your extracted features, outputs whether a light is red, green or yellow as a one-hot encoded label. This classification function should be able to classify any image of a traffic light!

You are encouraged to write any helper functions or visualization code that you may need, but for testing the accuracy, make sure that this `estimate_label` function returns a one-hot encoded label.

(IMPLEMENTATION): Build a complete classifier

```
In [27]: def sumNonZeroPixels(rgb_image):
             rows, cols, _ = rgb_image.shape
             count = 0

             for row in range(rows):
                 for col in range(cols):
                     pixel = rgb_image[row, col]
                     if sum(pixel) != 0:
                         count = count + 1
             return count
```

```
In [51]: # This function should take in RGB image input
         # Analyze that image using your feature creation code and output a one-hot encoded labe
         def estimate_label(rgb_image):

             hsv = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2HSV)
             sum_saturation = np.sum(hsv[:,:,1]) # Sum the brightness values
             area = 32*32
             avg_saturation = sum_saturation / area # Find the average

             # I don't know why but when I run the create_feature as below, the accuracy drops t
             # The above code is the same code written in create_feature.
             #avg_saturation = create_feature(rgb_image)


             sat_low = int(avg_saturation)
             val_low = 140

             # Red
             lower_red = np.array([150,sat_low,val_low])
             upper_red = np.array([180,255,255])
             red_mask = cv2.inRange(hsv, lower_red, upper_red)
             red_result = cv2.bitwise_and(rgb_image, rgb_image, mask = red_mask)

             # Yellow
             lower_yellow = np.array([10,sat_low,val_low])
             upper_yellow = np.array([60,255,255])
             yellow_mask = cv2.inRange(hsv, lower_yellow, upper_yellow)
             yellow_result = cv2.bitwise_and(rgb_image, rgb_image, mask = yellow_mask)

             # Green
```

```
    lower_green = np.array([70,sat_low,val_low])
    upper_green = np.array([100,255,255])
    green_mask = cv2.inRange(hsv, lower_green, upper_green)
    green_result = cv2.bitwise_and(rgb_image, rgb_image, mask = green_mask)


    sum_green = sumNonZeroPixels(green_result)
    sum_yellow = sumNonZeroPixels(yellow_result)
    sum_red = sumNonZeroPixels(red_result)

    if sum_red >= sum_yellow and sum_red >= sum_green:
        return one_hot_encode("red")# Red
    if sum_yellow >= sum_green:
        return one_hot_encode("yellow")# Yellow
    return one_hot_encode("green") # Green
```

4.1 Testing the classifier

Here is where we test your classification algorithm using our test set of data that we set aside at the beginning of the notebook! This project will be complete once you've pogrammed a "good" classifier.

A "good" classifier in this case should meet the following criteria (and once it does, feel free to submit your project): 1. Get above 90% classification accuracy. 2. Never classify a red light as a green light.

4.1.1 Test dataset

Below, we load in the test dataset, standardize it using the `standardize` function you defined above, and then **shuffle** it; this ensures that order will not play a role in testing accuracy.

```
In [52]: # Using the load_dataset function in helpers.py
         # Load test data
         TEST_IMAGE_LIST = helpers.load_dataset(IMAGE_DIR_TEST)

         # Standardize the test data
         STANDARDIZED_TEST_LIST = standardize(TEST_IMAGE_LIST)

         # Shuffle the standardized test data
         random.shuffle(STANDARDIZED_TEST_LIST)
```

4.2 Determine the Accuracy

Compare the output of your classification algorithm (a.k.a. your "model") with the true labels and determine the accuracy.

This code stores all the misclassified images, their predicted labels, and their true labels, in a list called `MISCLASSIFIED`. This code is used for testing and *should not be changed*.

```
In [53]: # Constructs a list of misclassified images given a list of test images and their label
         # This will throw an AssertionError if labels are not standardized (one-hot encoded)
```

```
def get_misclassified_images(test_images):
    # Track misclassified images by placing them into a list
    misclassified_images_labels = []

    # Iterate through all the test images
    # Classify each image and compare to the true label
    for image in test_images:

        # Get true data
        im = image[0]
        true_label = image[1]
        assert(len(true_label) == 3), "The true_label is not the expected length (3)."

        # Get predicted label from your classifier
        predicted_label = estimate_label(im)
        assert(len(predicted_label) == 3), "The predicted_label is not the expected len

        # Compare true and predicted labels
        if(predicted_label != true_label):
            # If these labels are not equal, the image has been misclassified
            misclassified_images_labels.append((im, predicted_label, true_label))

    # Return the list of misclassified [image, predicted_label, true_label] values
    return misclassified_images_labels


# Find all misclassified images in a given test set
MISCLASSIFIED = get_misclassified_images(STANDARDIZED_TEST_LIST)

# Accuracy calculations
total = len(STANDARDIZED_TEST_LIST)
num_correct = total - len(MISCLASSIFIED)
accuracy = num_correct/total

print('Accuracy: ' + str(accuracy))
print("Number of misclassified images = " + str(len(MISCLASSIFIED)) +' out of '+ str(to
```

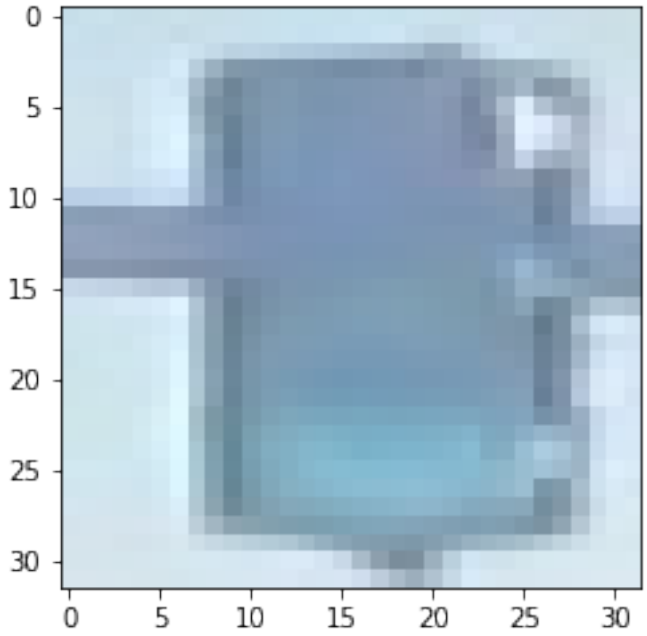
```
Accuracy: 0.9696969696969697
Number of misclassified images = 9 out of 297
```

Visualize the misclassified images

Visualize some of the images you classified wrong (in the `MISCLASSIFIED` list) and note any qualities that make them difficult to classify. This will help you identify any weaknesses in your classification algorithm.

```
In [31]: # Visualize misclassified example(s)
         ## TODO: Display an image in the `MISCLASSIFIED` list
         ## TODO: Print out its predicted label - to see what the image *was* incorrectly classi
         num = 1
         test_mis_img = MISCLASSIFIED[num][0]
         plt.imshow(test_mis_img)
         print(str(MISCLASSIFIED[num][1]))
```

```
[1, 0, 0]
```



(Question 2): After visualizing these misclassifications, what weaknesses do you think your classification algorithm has? Please note at least two.

Answer: This classifier does not handle very well images with fog, very white images or distorted width/height.

4.3 Test if you classify any red lights as green

To pass this project, you must not classify any red lights as green! Classifying red lights as green would cause a car to drive through a red traffic light, so this red-as-green error is very dangerous in the real world.

The code below lets you test to see if you've misclassified any red lights as green in the test set. **This test assumes that** `MISCLASSIFIED` **is a list of tuples with the order: [misclassified_image, predicted_label, true_label].**

Note: this is not an all encompassing test, but its a good indicator that, if you pass, you are on the right track! This iterates through your list of misclassified examples and checks to see if any red traffic lights have been mistakenly labelled [0, 1, 0] (green).

```
In [23]: # Importing the tests
         import test_functions
         tests = test_functions.Tests()

         if(len(MISCLASSIFIED) > 0):
             # Test code for one_hot_encode function
             tests.test_red_as_green(MISCLASSIFIED)
         else:
             print("MISCLASSIFIED may not have been populated with images.")
```

TEST PASSED

5 5. Improve your algorithm!

Submit your project after you have completed all implementations, answered all questions, AND when you've met the two criteria: 1. Greater than 90% accuracy classification 2. No red lights classified as green

If you did not meet these requirements (which is common on the first attempt!), revisit your algorithm and tweak it to improve light recognition -- this could mean changing the brightness feature, performing some background subtraction, or adding another feature!

5.0.1 Going Further (Optional Challenges)

If you found this challenge easy, I suggest you go above and beyond! Here are a couple **optional** (meaning you do not need to implement these to submit and pass the project) suggestions: * (Optional) Aim for >95% classification accuracy. * (Optional) Some lights are in the shape of arrows; further classify the lights as round or arrow-shaped. * (Optional) Add another feature and aim for as close to 100% accuracy as you can get!

```
In [ ]:
```