Vehicle Detection Project Report

Definition

Humans. We are highly social and emotional creatures. With repetitive tasks, we often get bored, and our efficiency begins to reduce when we work for long hours. We don't like to be confined to something for very long.

With the growing size of the world population and the rising demand for better and faster services, there is a need for faster and more efficient ways of doing things. What better way than to use machines for the job!? And that's precisely why automation exists.

Modern machines are highly complex tools that have made automation possible. Automation has changed the way people work and live. It helps us with tasks that are often repetitive, boring or even dangerous in nature—so that we may spend our time in more meaningful pursuits. With the advent of industrial revolution, the steam engine and ultimately the computers—laborious tasks which used to require many hours of human effort are now being done much more efficiently and quickly by machines.

The research and implementation of **autonomous vehicles**, in particular, has an immense impact in our everyday lives. According to wikipedia—An autonomous vehicle, or a self driving car is a vehicle that is capable of sensing its environment and navigating without human input. Just like any other machine, a self-driving car has many components that work together to transport people and goods safely from one place to another.

Autonomous vehicles have the potential to transport people and goods much more safely than human drivers, and also on a much larger scale. There are plenty of challenges and problem areas that need to be addressed in order to achieve this objective. With the latest advances in computing power, it's become possible for us to implement various mathematical and scientific research in solving these problems.

Problem Statement

It's important to ensure that an autonomous vehicle detects the presence of other vehicles on the road—so that it does not pose a danger to itself or to other vehicles around it. In this Capstone project, we explore an area that is paramount to ensuring safety in autonomous vehicles—**computer vision**.

Evaluation Metrics

We divide the dataset into training and testing subsets. Then, we'll use a technique called Histogram of Oriented Gradients (HOG) to extract features from our training data, and train a SVM classifier (Suppport Vector Machines).

Once we train our model using the data from the training set, we can measure the accuracy of our model using images from the validation set. We can experiment with tuning different hyper-parameters to get the best accuracy for our model.

Analysis

Data Exploration

We use two different sets of data for our purposes. The first data-set consists of vehicle images, and the other data-set consists of non-vehicular objects on the road. We need both these data-sets so that our agent can learn to distinguish between vehicles and non-vehicles on the road.

Exploratory Visualization

All the training images are of size 64x64 pixels.

For recognizing vehicles, the vehicle data-set has 4 different categories of images which show vehicles ahead of our agent from different viewing angles and distances. These images will be used to train our agent to recognize vehicles towards its left side and right side, vehicles that are far, and the ones that are closer.

The second data-set consists of images of objects other than vehicles that can be seen on the road. These could include trees, pavement markings, side-walks, traffic signs, road dividers, the sky in various times of the day, and so forth.

Algorithms and Techniques

First, we'll begin with feature exploration. The process begins with extracting useful information from the image and eliminating noise data, thus making the image simpler. The resulting image is called Feature Descriptor. HOG is a type of feature descriptor that is typically used in our given objective.

Here are the steps to calculate a HOG descriptor—

- We first calculate the horizontal and vertical gradients in the given image.
- Next, we find the magnitude and direction of each gradient using this formula

$$g = \sqrt{g_x^2 + g_y^2}$$

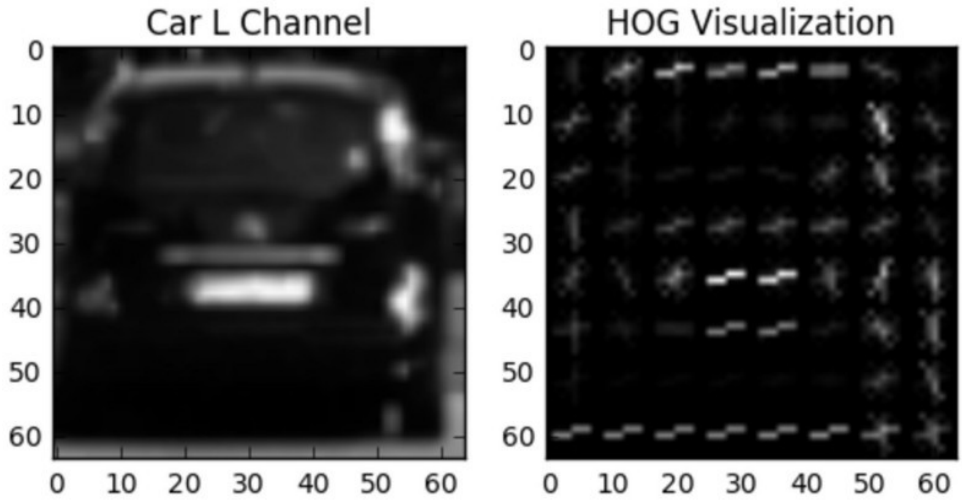
$$\theta = \arctan \frac{g_y}{g_x}$$

This removes a lot of noisy information from the image, but highlights all the essential outlines.

- Now calculate a Histogram of Gradients by dividing the image by x*y cells
- Finally we normalize the gradient vectors, so that the training becomes independent of the brightness of the image, contrast etc.

Using our selected HOG features, we train an SVM classifier. We'll experiment with parameters like the type of kernel used, gamma, color channels etc, and use GridSearchCV to find the optimum hyper-parameter tuning.

If we take a look at a single color channel for a random car image, and its corresponding HOG visulization, they look like this:



Next, we'll use a sliding window search to try and detect vehicles in a live video recording taken from a moving car.

Methodology

Data Preprocessing

Spatial Binning:

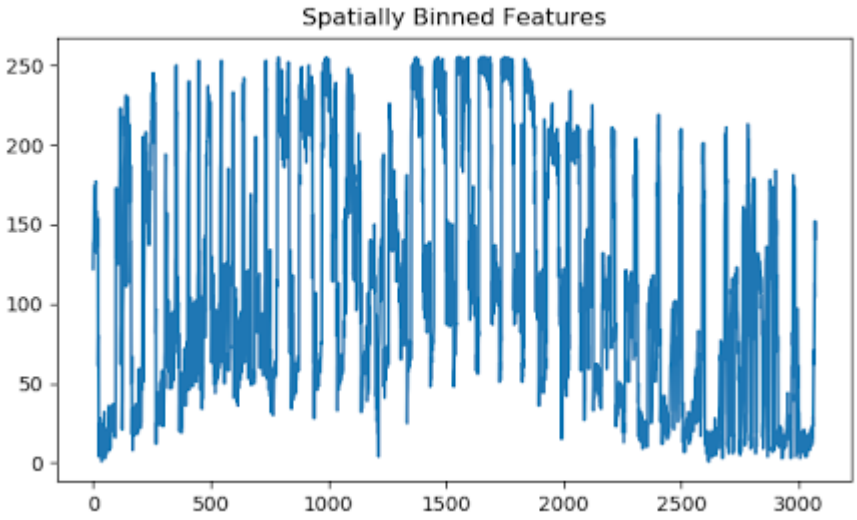
Template matching is not a particularly robust method for finding vehicles unless you know exactly what your target object looks like. However, raw pixel values are still quite useful to include in your feature vector in searching for cars.

While it could be cumbersome to include three color channels of a full resolution image, you can perform spatial binning on an image and still retain enough information to help in finding vehicles.

As you can see in the example above, even going all the way down to 32 x 32 pixel resolution, the car itself is still clearly identifiable by eye, and this means that the relevant features are still preserved at this resolution.

A convenient function for scaling down the resolution of an image is OpenCV's cv2.resize(). You can use it to scale a color image or a single color channel, this pre-processing step is important beforewe train our classifier.

And the output would look someting like this -



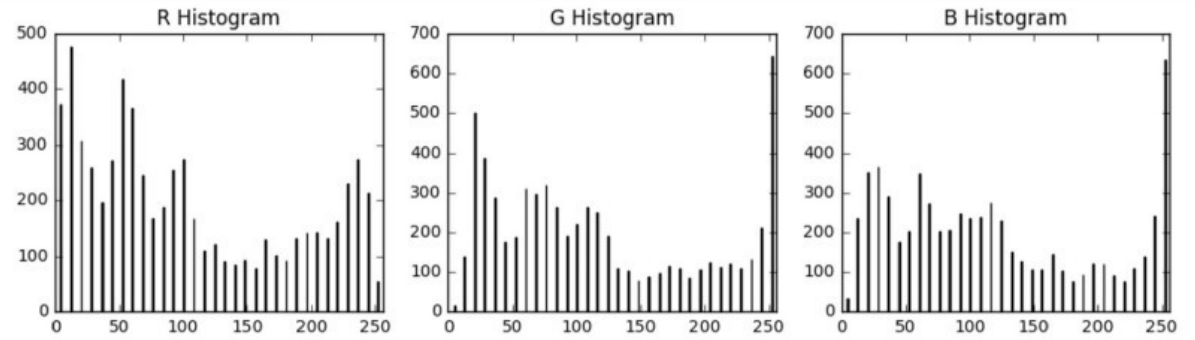
Implementation

- **Color Histograms:**

An image template is useful for detecting things that do not vary much in their appearance—for example, icons of emojis. But for most real world objects that do appear in different forms, orientation, and sizes, this technique does not work quite well. In template matching, you depend on raw color values laid out in a specific order, and that can vary a lot. So you need to find some transformations that are robust to changes in appearance. One such transform is to compute a histogram of color values for an image.

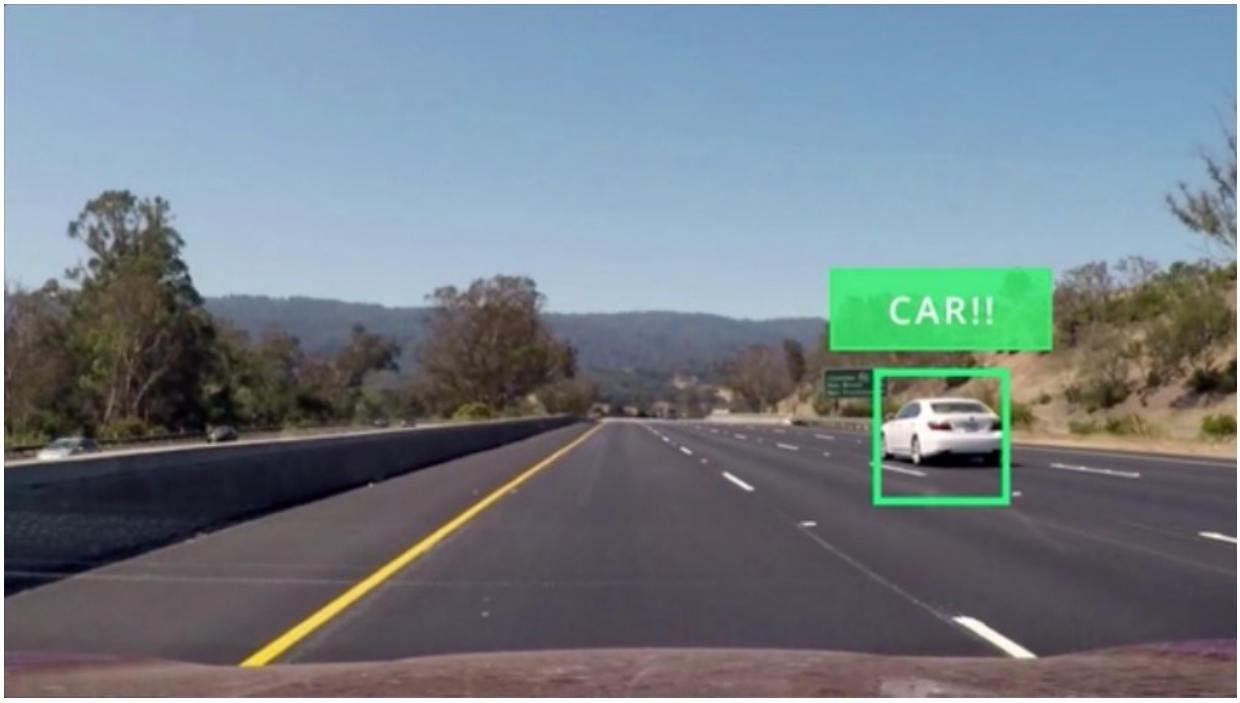
When you compare the histogram of a known object with the regions of a test image, locations with a similar color distribution will reveal a close match. So we are no longer sensitive to a perfect arrangement of pixels. So objects that appear in slightly different orientations and sizes will still be a match.

This is how the color histograms look like:



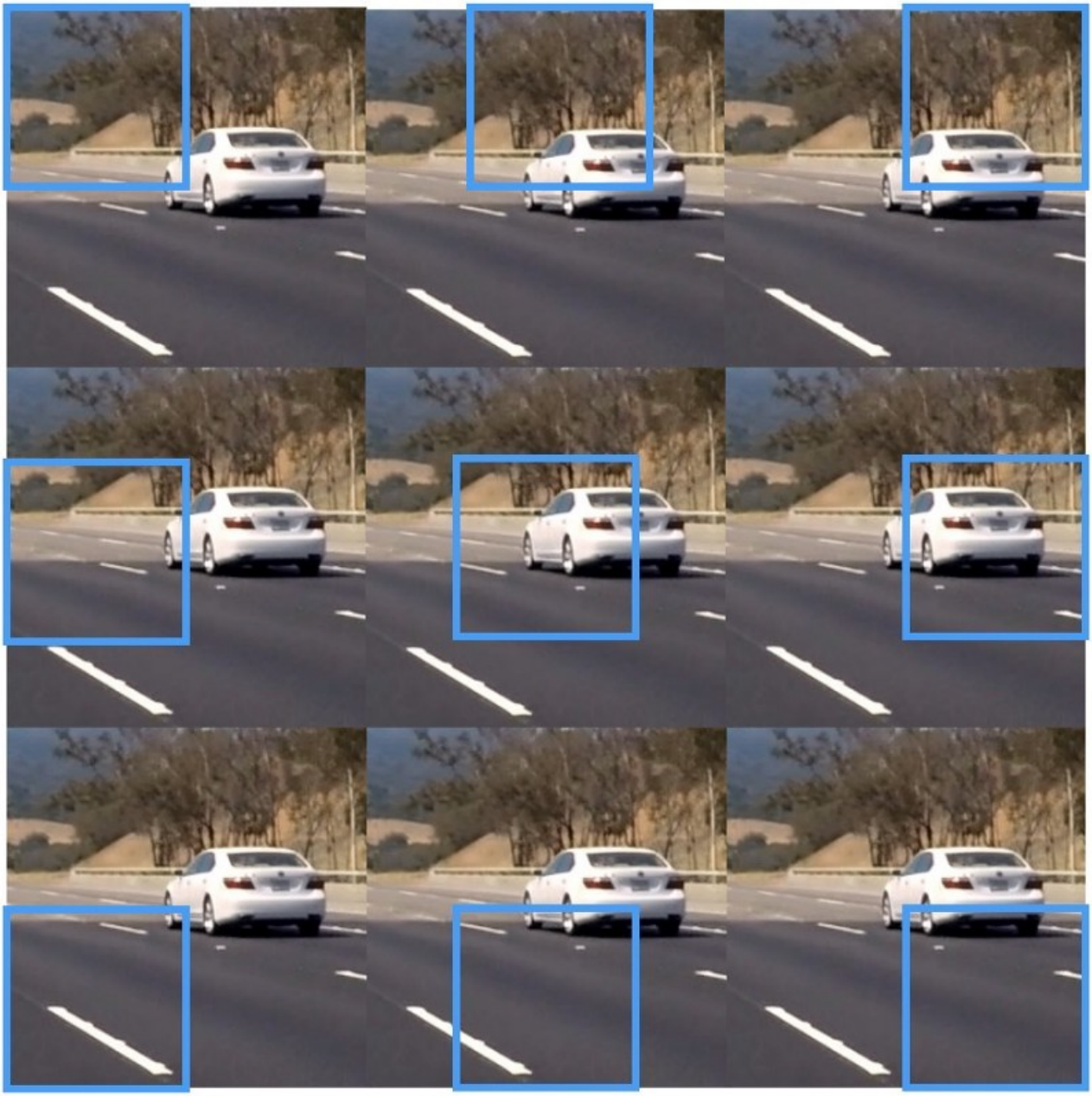
These, collectively, are now our feature vector for this particular cutout image.

- **Sliding Window Search**



To implement a sliding window search, you need to decide what size window you want to search, where in the image you want to start and stop your search, and how much you want windows to overlap. So, let's try an example to see how many windows we would be searching given a particular image size, window size, and overlap.

Suppose you have an image that is 256 x 256 pixels and you want to search windows of a size 128 x 128 pixels each with an overlap of 50% between adjacent windows in both the vertical and horizontal dimensions. Your sliding window search would then look like this:



The goal here is to write a function that takes in an image, start and stop positions in both x and y (imagine a bounding box for the entire search region), window size (x and y dimensions), and overlap fraction (also for both x and y).

Refinement

We use the Histogram of Gradients technique and refine it further to get our desired results.

The scikit-image package has a built in function to extract Histogram of Oriented Gradient features. The documentation for this function can be found hereand a brief explanation of the algorithm and tutorial can be found here.

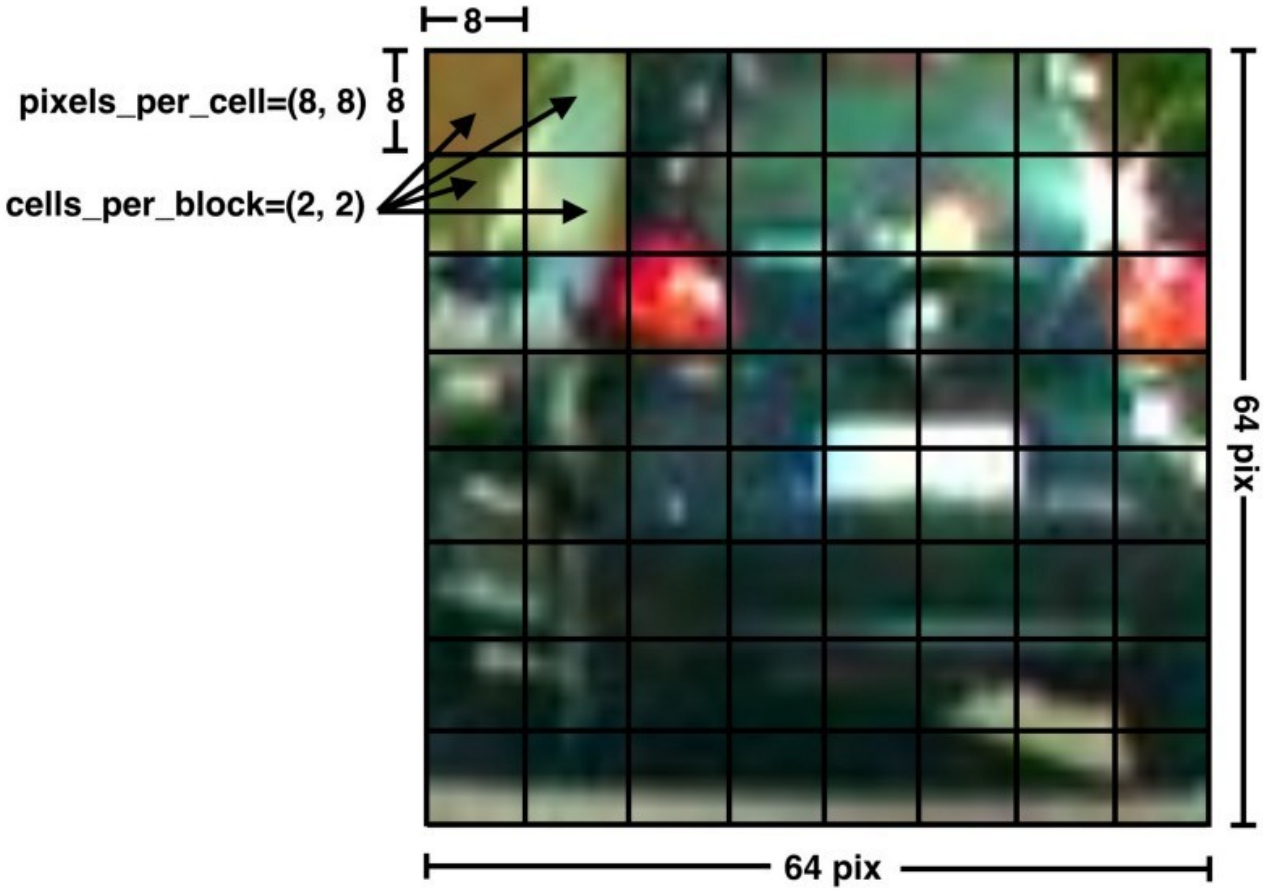
The scikit-image *hog()* function takes in a single color channel or grayscaled image as input, as well as various parameters. These parameters include orientations, pixels_per_cell and cells_per_block.

The *number of orientations* is specified as an integer, and represents the number of orientation bins that the gradient information will be split up into in the histogram. Typical values are between 6 and 12 bins.

The *pixels_per_cell* parameter specifies the cell size over which each gradient histogram is computed. This paramater is passed as a 2-tuple so you could have different cell sizes in x and y, but cells are commonly chosen to be square.

The *cells_per_block* parameter is also passed as a 2-tuple, and specifies the local area over which the histogram counts in a given cell will be normalized. Block normalization is not necessarily required, but generally leads to a more robust feature set.

There is another optional power law or “gamma” normalization scheme set by the flag transform_sqrt. This type of normalization may help reduce the effects of shadows or other illumination variation, but will cause an error if your image contains negative values (because it’s taking the square root of image values).



This is where things get a little confusing though. Let’s say you are computing HOG features for an image like the one shown above that is 64×64 pixels. If you set pixels_per_cell=(8, 8) and cells_per_block=(2, 2) and orientations=9. How many elements will you have in your HOG feature vector for the entire image?

You might guess the number of orientations times the number of cells, or 9×8×8=576, but that’s not the case if you’re using block normalization! In fact, the HOG features for all cells in each block are computed at each block position and the block steps across and down through the image cell by cell.

So, the actual number of features in your final feature vector will be the total number of block positions multiplied by the number of cells per block, times the number of orientations, or in the case shown above: 7×7×2×2×9=1764.

Results

Model Evaluation and Validation

- **SVM Hyperparameters**

We can optimize the Gamma and C parameters for an SVC classifier.

Successfully tuning your algorithm involves searching for a kernel, a gamma value and a C value that minimize prediction error. To tune your SVM vehicle detection model, you can use one of scikit-learn's parameter tuning algorithms.

When tuning SVM, remember that you can only tune the C parameter with a linear kernel. For a non-linear kernel, you can tune C and gamma.

- **Parameter Tuning in Scikit-learn**

Scikit-learn includes two algorithms for carrying out an automatic parameter search:

- GridSearchCV
- RandomizedSearchCV

GridSearchCV exhaustively works through multiple parameter combinations, cross-validating as it goes. The beauty is that it can work through many combinations in only a couple extra lines of code.

For example, if I input the values C:[0.1, 1, 10] and gamma:[0.1, 1, 10], gridSearchCV will train and cross-validate every possible combination of (C, gamma): (0.1, 0.1), (0.1, 1), (0.1, 10), (1, .1), (1, 1), etc.

RandomizedSearchCV works similarly to GridSearchCV except that it takes a random sample of parameter combinations. It's faster than GridSearchCV since it uses a subset of the parameter combinations.

- **Cross-validation with GridSearchCV**

GridSearchCV uses 3-fold cross validation to determine the best performing parameter set. GridSearchCV will take in a training set and divide the training set into three equal partitions. The algorithm will train on two partitions and then validate using the third partition. Then GridSearchCV chooses a different partition for validation and trains with the other two partitions. Finally, GridSearchCV uses the last remaining partition for cross-validation and trains with the other two partitions.

By default, GridSearchCV uses accuracy as an error metric by averaging the accuracy for each partition. So for every possible parameter combination, GridSearchCV calculates an accuracy score. Then GridSearchCV will choose the parameter combination that performed the best.

Justification

- I used the validation score of the machine learning model on a sample of the data set along with GridSearchCV to predict the best set of parameters for HOG.
- The accuracy scored the best with `YCrCb` color space.
- After trying various combinations of features against the trained model, the prediction accuracy was its best for the following paramters, and these were chosen as final set of parameters.

```
colorspace = 'YCrCb'
orient = 8
pix_per_cell = 8
cell_per_block = 2
hog_channel = 'ALL'
spatial_size = (16, 16)
hist_bins = 32
hist_range=(0, 256)
```

Conclusion

Free Form Visualization

I've provided the final output video in the file named *video-tracking-output.mp4.*

Reflection

This was a quite challenging problem. One specific part where I found difficulty was when I had to scale down the size of the images—because the classifer was trained on image samples of 64x64 pixels. Applying a Scaler transform helped fix this.

Using GridSearchCV helped with some of the challenges, especially when considering the optimum color channel. Prediction on a video also takes significant time, owing to the significantly large number of frames in the video.

Overall, this was a very interesting project.