Submitted by Kyle Martin

# Project 5
# Vehicle Detection and Tracking

The goals / steps of this project are the following:

- Perform a Histogram of Oriented Gradients (HOG) feature extraction on a labeled training set of images and train a classifier Linear SVM classifier
- Apply a color transform and append binned color features, as well as histograms of color, to the HOG feature vector
- Implement a sliding-window technique and use a trained classifier to search for vehicles in images.
- Run the pipeline on a video stream and create a heat map of recurring detections frame by frame to reject outliers and follow detected vehicles.
- Estimate a bounding box for vehicles detected.

Submitted by Kyle Martin

## Feature Extraction

The approach I decided to take in feature extraction was to combine a Histogram of Gradient and YCrCb histogram into a single feature vector.

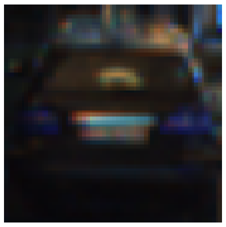
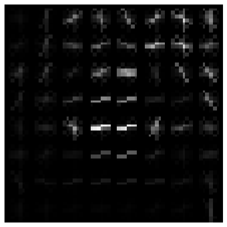
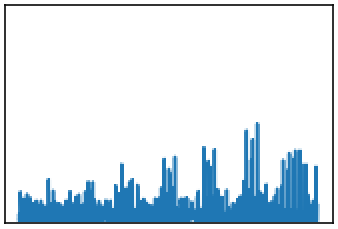
| Parameter | Value |
|---|---|
| Orientation Bins | 9 |
| Pixels per Cell | 8 |
| Cells per Block | 2 |
| Spatial Bins | 32x32 |
| Histogram Bins | 32 |

### Histogram of Gradients

Histogram of Gradients is a shape feature technique that essentially creates a fingerprint of shape. This fingerprint can be used later to identify if a complex shape exists in an image.

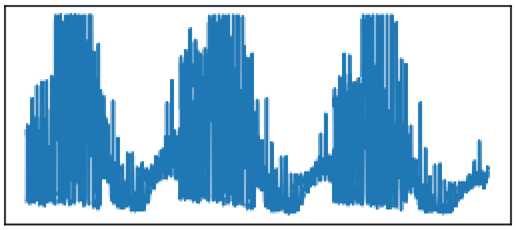
I created a class called FeatureExtractor to encapsulate all of the feature detection functionality in this project. To extract the HOG I used skimage's HOG function. The code that utilized this function is in feature_extractor.py line 11 - 29.

The parameters for the HOG algorithm were determined by testing many different values and optimizing for good detection and low false-positives.

### Color Histogram

Another way to fingerprint an image is by looking at the mix of colors in it. In this project I chose to convert each frame into the 3-channel YCrCb colorspace as it does a good job capturing how the human eye sees differences in color. I then take a 32 binned histogram of each channel and concatenate them into a single feature vector.



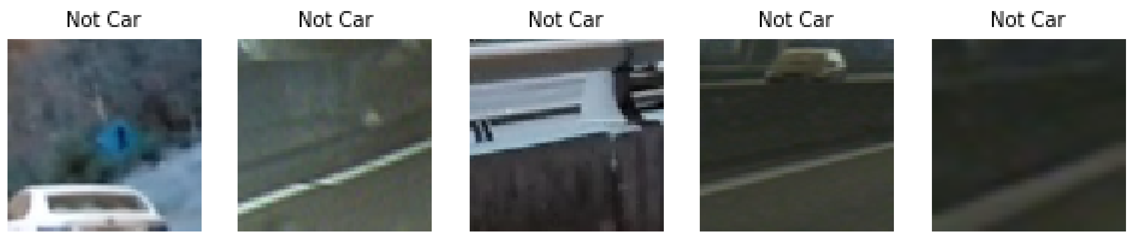
## Classifying A Car

### Data

Now that we have feature vector, HOG + Color Histogram, we can build a classifier to robustly identify cars in an image. I chose to used a SVM as the classifier and put the training code in an iPython notebook called TrainSVM.ipynb

Next I collected a bunch of training images from Udacity, GTI, and KITTI. Below are some random images with their label.

| | |
|---|---|
| Car Class Examples | 8792 |
| Not Car Class Examples | 9666 |



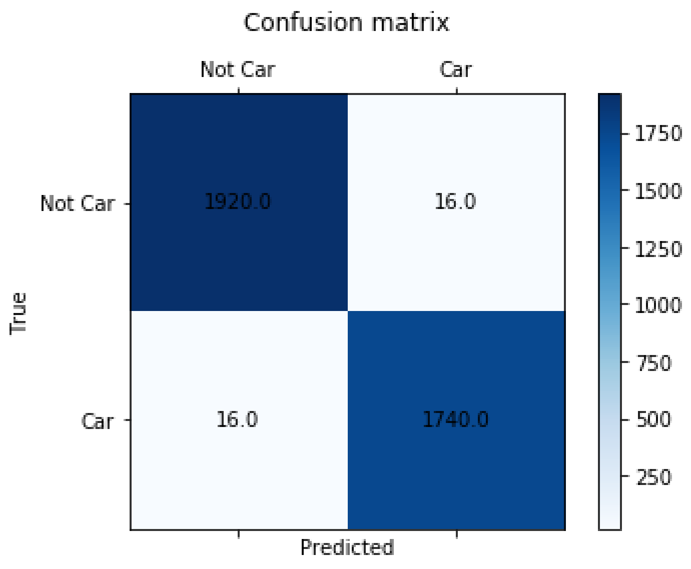
## Training

I trained the SVM using a feature vector comprised of the HOG of the image + color histogram of each channel of the image. The final feature vector ended up being 8460 bytes long. The code for training the SVM is located in TrainSVM.ipynb section 70.

## Classification Performance

After the model is trained, it’s time to measure how accurate it is. To do this I used several metrics built into the sklearn.metrics package. These metrics are are laid out below. Overall, the accuracy ended up being 99.13% accurate.

| | |
|---|---|
| Accuracy Score | 0.9913 |
| F1 Score | 0.9913 |
| Precision Score | 0.9913 |
| Recall Score | 0.9913 |



The code for this is in TrainSVM.ipynb sections 71-72.

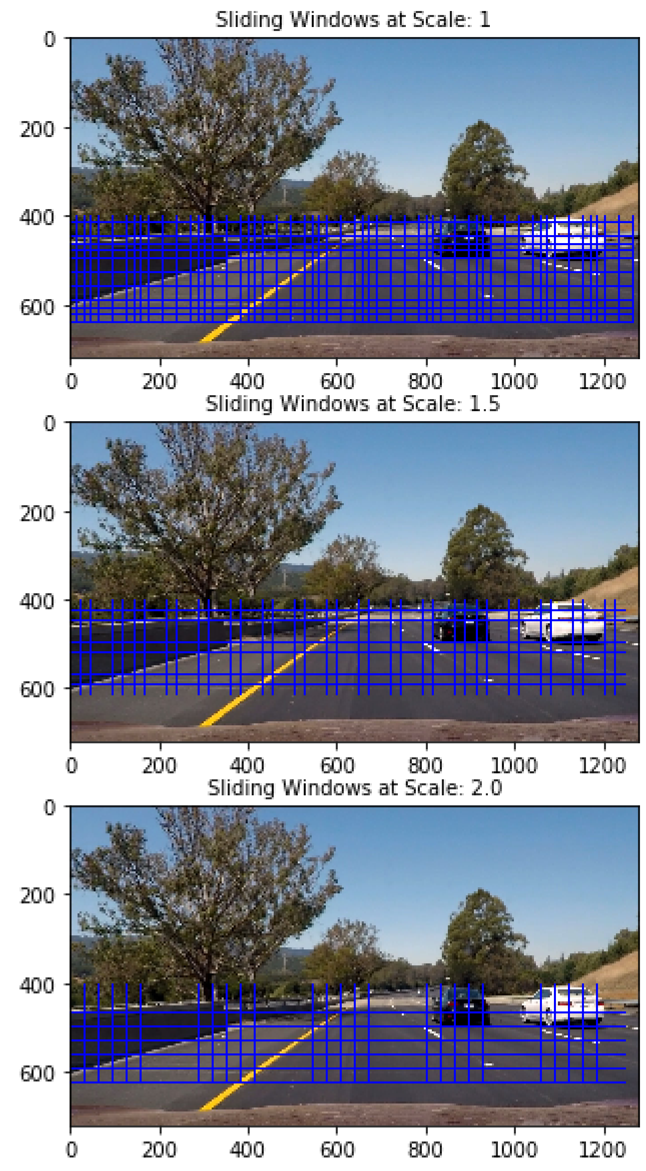
## Sliding Window Search

Now that we can tell if a car is in an image or not, it's time to start feeding the classifier slices of video frames and start to tracking cars.

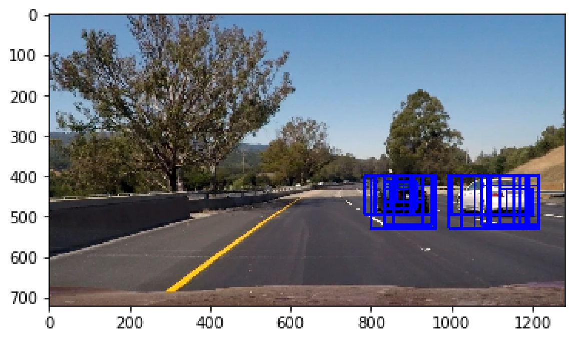
### Implementation

In order to search the image for the cars, we first make 3 different scales of the original frame at 1x, 1.5x smaller and 2.0x smaller. The reason for this is that as the cars get really far from the camera, their gradients values don't match what the classifier was trained on. So we must make a few smaller versions and search these too. This code is located at car_finder.py lines 102-122.

Once the different image scales are made then it's time to slice them up into patches and scan for a car. Each patch is turned into a feature vector and passed in the SVM classifier.


Sliding Windows at Scale: 1
Sliding Windows at Scale: 1.5
Sliding Windows at Scale: 2.0
0
200
400
600
800
1000
1200

Every time a patch does contain car we save the location and size of that patch to an array.



This code is located in car_finder.py lines 124-203.

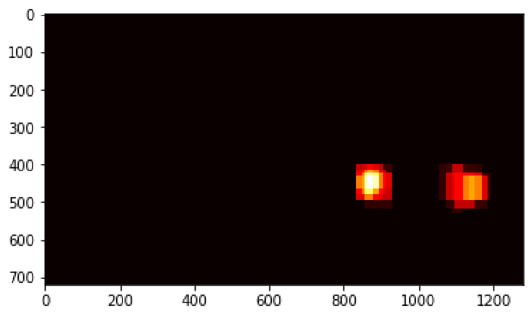
## Video Implementation

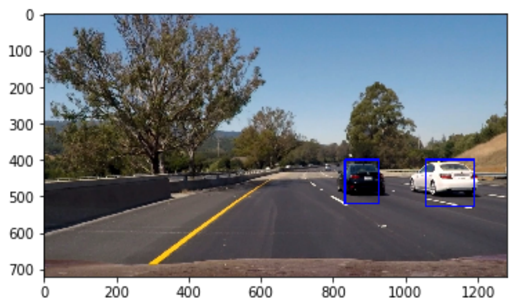
A video of this tracking algorithm can be found at:
https://www.youtube.com/watch?v=XDu99D_XHPE

### Filtering

As you can see above, there are clusters of bounding boxes around each car. To filter noise and find the average of these clusters I chose to create a discrete heatmap. This heatmap is thresholded to remove noise and I then use sklearn to find the bounding boxes of each hot spot. These hotspots correspond to a car or multiple cars, depending how close they car.

Below is an example of the heatmap of the bounding boxes and then a final bounding drawn on the original video frame.





The code for this is located in car_finder.py lines 54-95.

## Conclusion

Overall, I really enjoyed this project. It gave me even more insight into computer vision techniques and an introduction to SVM image classification.

It was not without it’s challenges though. There are so many parameters to tune so one can certainly go down a rabbit hole.

There are a lot of ways you could make this algorithm more robust. First, I’d add more training data to train the SVM with. Images with weird perspectives would really help the classifier. Second, I also add a deep neural network to the car search phase to provide a second opinion on the matter. Third, I’d create a system to predict each car’s trajectory with an optical flow tracking algorithm. If you can predict each car’s trajectory then you can begin to plan possible safes path on the road ahead.