Udacity SDC – P5: Vehicle Detection & Tracking Write-up Report

By: James Beasley

In this report, I will consider the rubric points individually and describe how I addressed each point in my implementation.

Submission Contents & Execution

My project includes the following directories/files:

- **model_training** directory containing data sets, pickled objects, training and test set preparation code, feature extraction code, and model training code
- **output_images** directory containing a final output image from the pipeline (it was difficult to auto-generate each stage's output to this directory, so I took snapshots of the stages and put them in the writeup report
- **output_video** directory containing final output video: *processed_project_video.mp4*
- **sdcp4** directory containing all dependencies from project 4
- **test_images** directory containing Udacity-provided pipeline test images (and a few I exported from the project video)
- **test_video** directory containing Udacity-provided project videos: *project_video.mp4* and *test_video.mp4*
- **main.py** containing initialization, test (export final output image), and production (export processed video) pipeline execution code
- **production_pipeline.py** containing code that processes each frame of the project video through my pipeline and produces a video saved to the output_video directory
- **test_pipeline.py** containing code that saves final output image from the pipeline to the output_images directory
- **vehicle_processor.py** containing code that performs a vehicle search using a sliding window technique and a trained classifier, also thresholding code to deal with false positives and object labeling using heatmaps
- **writeup_report.pdf** summarizing the results (this document)

To execute: first, clone the project from: https://github.com/embeddedcognition/sdcp5.git

Once cloned, an example image of the final stage of my pipeline is auto-generated and saved in the **output_images** directory and a processed version of the provided project video is also auto-generated and saved in the **output_video** directory by executing:

```
python main.py
```

Overview

I have incorporated my vehicle detection and tracking process into the pipeline I delivered in the advanced lane finding project (please refer to my detailed write-up from P4 for details on its inner-workings). I won't revisit those portions but will address the deltas I've added to meet the requirements of this current project.

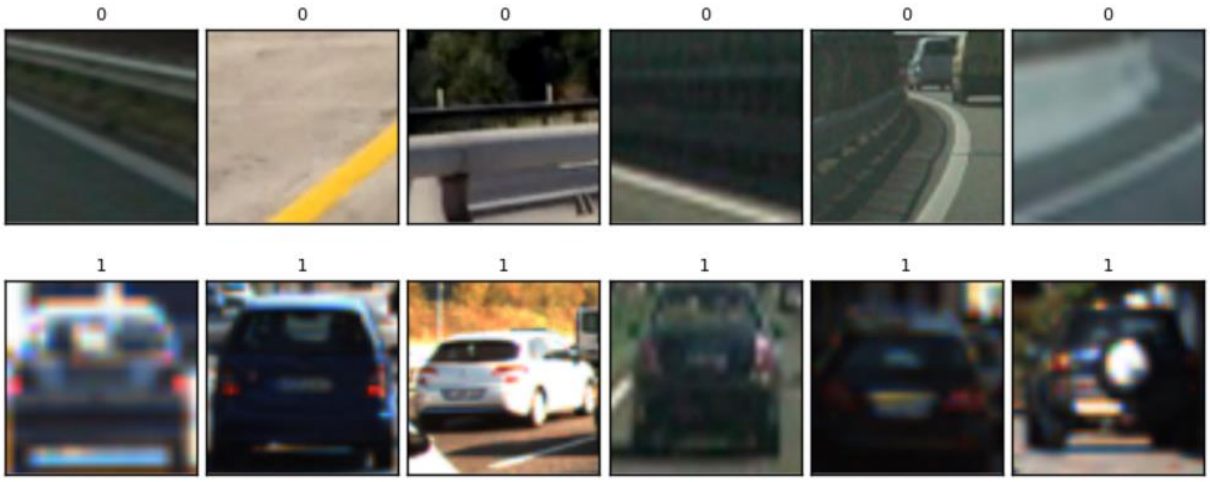
As before, my submission includes two pipelines: The first is the test pipeline (**test_pipeline.py**) which saves an example of the final output of the pipeline to the **output_images** directory (see lines: 112–152 related to this current project). The second is the production pipeline (**production_pipeline.py**) which processes each frame of the Udacity-provided project video (*project_video.mp4*) through each stage (0-part-2 through 4 of my P4 submission) and the newly added vehicle detection and tracking capability I've added with P5 (see lines: 28–41, 61–67, 183–209). Then finally, it saves a processed video (processed_*project_video.mp4*) to the **output_video** directory. I'll describe the sequence of processing of the data preparation, model training, and finally the pipeline processing the project video.

1. Training and Test Data Set Creation

First off, the project data sets need to be downloaded from my AWS S3 bucket. There are links and instructions in the readme files contained in the "*model_training/data_sets/vehicles*" and "*model_training/data_sets/non_vehicles*" directories to accomplish this (**note:** you only need to do this if you want to train the model from scratch). Essentially you just need to extract each zip file under its relevant path. You then need to execute the **"model_training/train_test_data_processor.py"** file (via *python train_test_data_processor.py*) to prepare a training set and test set. Each is pickled and saved in the "*model_training/pickled_objects*" directory.

The "vehicle" data set (provided by Udacity) contains 8,792 images and the "non-vehicle" data set (also provided by Udacity) contains 8,968 images. Below are some example images from them:

```
Six random images from each class (and its label above) are shown below:
```



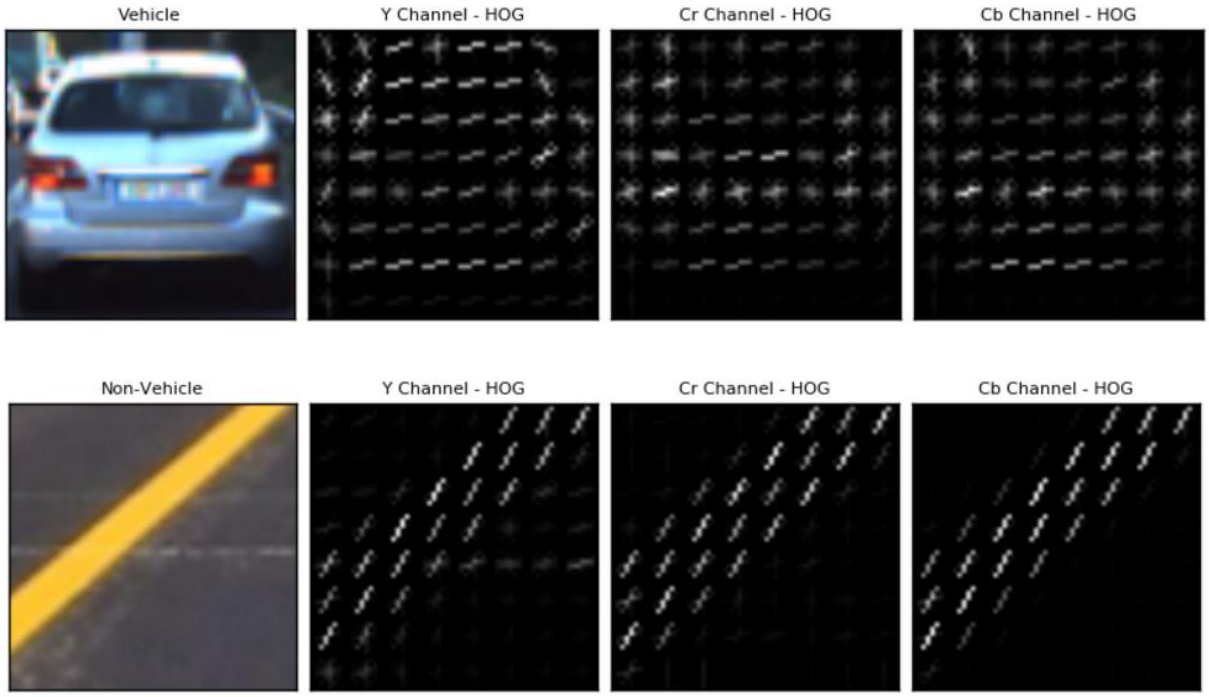

```
Number of class instances by class:
Counter({0: 7174, 1: 7034})

Majority class: 0
Majority class count: 7174
```

2. Feature Extraction & Model Training

I'll first discuss my feature extraction strategy. During early experimentation, I found that an ensemble approach to feature construction (leveraging raw pixel intensities (targeting color and shape) and gradient of raw pixel intensities (targeting shape)) was key to achieving strong results in the task of classification. When using any of them alone or in part, I wasn't able to achieve a test set accuracy of more than 94%. But together, I was able to achieve 99.4%.

My feature extraction code is all contained within the **"model_training/feature_processor.py"** file. Lines 18–33 perform HOG feature extraction from an image channel, allowing the model to learn shape, lines 35–38 perform spatial reduction (binning) on an image, allowing the model to learn color and shape characteristics at lower resolution (and cost), and lines 40–47 compute the pixel intensity frequency distribution of an image to allow the model to learn about color variances.

The **"model_training/model_train_processor.py"** file contains the code to train and pickle the model and the scaler for use in the pipeline (it can be run via *python model_train_processor.py*, **note:** you only need to do this if you want to train the model from scratch, a pickled version of the trained model and scaler are included in this submission). Lines 24–50 perform feature extraction (as described earlier) on the training set. I found that the YCrCb color space (leveraging all three channels with HOG) achieved the best results, giving my model an accuracy improvement of 0.5 over other color spaces tried. This choice of color space definitely improved the shape definition in the HOG images seen below:



My strategy for classification was to follow the path of using a support vector machine, as I hadn't worked with them in the past and wanted to get some experience. As described in the lessons, I scaled my features (see lines 113–118 in model_train_processor.py), then used the LinearSVC from sklearn (see lines 127–134 in model_train_processor.py).

Regarding hyper-parameters (see lines 98–103 in model_train_processor.py), I found that making adjustments to the set of HOG variables (orientation-bins, pixels-per-cell, cells-per-block) didn't give me much improvement (they were already optimized ☺), so I decided to leave those as they were described in the lessons (as described earlier, the YCrCb color space choice was the most important factor to good HOG performance for me). For the set of raw pixel intensity variables (spatial-reduction-size, pixel-intensity-frequency-distribution-bins), I found that tweaking them had a big effect on my model performance. I found that any value below or above 64 for the pixel-intensity-frequency-distribution-bins kept me from achieving > 99% test set accuracy. Same for the spatial-reduction-size, when I retained less spatial information my model suffered. Though my final feature vector size was 8556, I felt it achieved great results.

My final model training results:

```
Test Accuracy of SVC =  0.9949
My SVC predicts:  [1 1 0 1 0 1 1 1 1 1]
For these 10 labels:  [1 1 0 1 0 1 1 1 1 1]
0.0026 Seconds to predict 10 labels with SVC
```

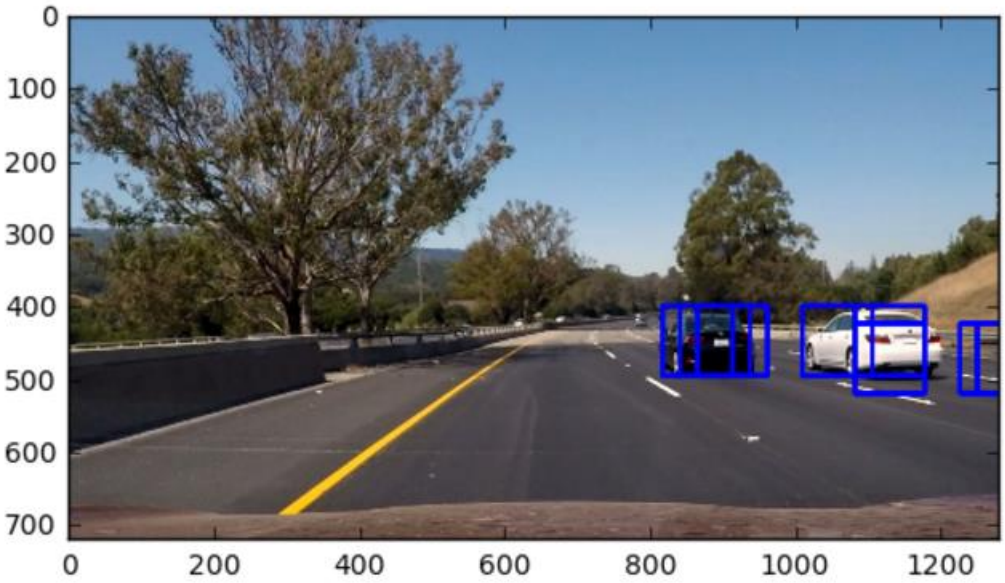
3. Pipeline & Sliding Window Strategy

The code for detecting and tracking vehicles is in the **vehicle_processor.py** file. For my sliding window strategy, I followed the example given in the HOG sub-sampling window search lesson, as I felt it would be much more efficient to compute the HOG features once for each scaled image vs. per window in that image. Beyond the few tweaks I made to return the set of positive prediction window coordinates (instead returning an image with them drawn on there), the code is largely the same as was provided in the lesson (see the *perform_vehicle_search* function, lines 60–137). This algorithm uses steps of 2 cells (~75% window overlap) vs. defining a specific overlap.

I chose to employ 4 window sizes to locate vehicles at multiple depths-of-field/orientations. My window scales are defined in the **production_pipeline.py** file at line 34. My window sizes are: 128x128 (64 pixels * scale of 2.0), 96x96 (64 pixels * scale of 1.5), 76x76 (64 pixels * scale of 1.2), 64x64 (64 pixels * scale of 1). These sizes/scales were arrived at through experimentation with test images. They did the best at identifying vehicles in different locations/conditions.

I also chose to employ the same y-axis cropping in the *perform_vehicle_search* function as was described in the lessons (i.e., 400 to 656 on the y-axis). Below is what the cropped frame looks like:

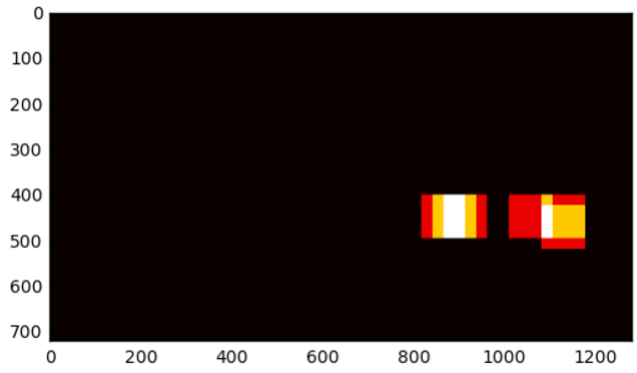
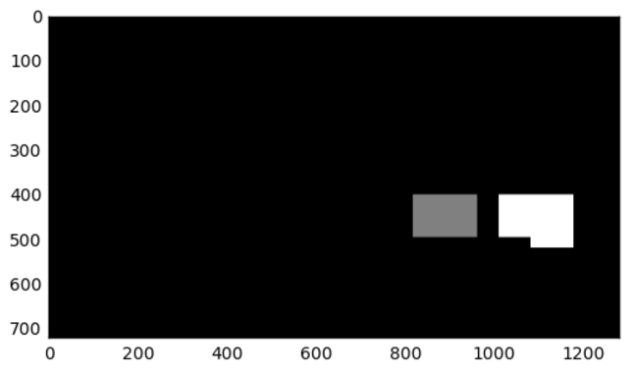
Below are the resulting detections from the *perform_vehicle_search* function drawn back on the original frame. You see that there are some false positives on the right-side edge.



So far, I had spent a lot of time improving the reliability of my classifier by employing an ensemble of features (as was discussed earlier) to ensure the trained model had good intuition around detecting vehicles in various conditions (employing multiple search window sizes helped as well). With that said, the model wasn't perfect, so I had to add additional intuition to improve the overall robustness of my pipeline to combat times with the model failed to predict correctly.

To combat the false positives we see above, I employed a decision function (see lines 124–130 of the *perform_vehicle_search* function) during the prediction process to improve overall confidence that the prediction was good. This helped reduce false positives a lot.

On top of that, I kept the positive predicted windows for each frame in a queue, 15 frames in total (see lines 192–206 of the production_pipeline.py file), and then for each frame, added all of those detections to a heatmap (defining the blobs that give us strong confidence regarding vehicle location), that I then thresholded (shown below) to get rid of noise (false positives). The heatmap really helped with the sanity check on where the objects should be over the course of several frames and cut out anomalies a lot. That result was then sent to an object labeling function (see lines 45–58 of the vehicle_processor.py file and 205–209 of the production_pipeline.py file) which made the final call on the shape/split of the resulting bounding boxes for the heatmap blobs.

Heatmap	Labeled Objects
	

4. Resulting Detected & Tracked Vehicles

Once the final bounding boxes are determined, they're drawn back on the frame. Example output of the P4 requirements + the detected/tracked vehicles for P5:

My completed video can be found at:

https://www.youtube.com/watch?v=UUSzSIlLoDEvq=hd720

Alternatively, it's located in the **output_video** folder of my submission, the file is named: **processed_project_video.mp4**

Discussion

I started off down an audacious path, using the project data set, but also adding in the Udacity data set 1 (just the bounded objects in each image), and then to balance the data set again, sampling from the minority class (non-vehicle) and creating synthetic data (using translations and brightness adjustment). This took many hours to put together but I was really learning a lot along the way. As I worked to tweak the model hyper-parameters, I hit a brick wall at 98.5% accuracy. No matter what I did, I couldn't get additional improvement. Additionally, when I tested the model on the test images it wasn't doing a great job of identifying the vehicles and there were lots of false positives. Given the amount of work I had left to do to complete the project, I decided to abandon this approach and go back to using the provided project data set (vehicles and non-vehicles) without any augmentation. This turned out to be a smart idea, as I made lots of progress quickly; training a model that achieved 99.4% accuracy on the test set and performed well on test images and the project video.

I really learned a lot during this project and feel like I was able to implement some interesting things (such as: linear svc that performed well on the test set, leveraging a decision function to improve confidence during the prediction process, heatmapping).

The biggest hurdle I faced on this project was tweaking all the hyper-parameters in a way that gave a good result. There were many combinations to explore and the toughest one was probably the combination of decision function threshold and heatmap threshold to deal with false positives, but also detect and track the vehicles appropriately. Sometimes extended experimentation is the best thing you can do. ☺

I think there is definitely room for improvement in my pipeline, especially in the area of performance. At best, it takes 45 minutes to process the full video (a good bit of this is due to my choice of 4 window sizes). I would have liked to explore more around how to get a good result in a performant manner but time got the best of me. I would also like to do more work to understand how I could improve the classifier as to not have to compensate so much with regard to false positive thresholding.

I will definitely be revisiting this codebase in the future!