Vehicle Detection System

Write up: Aneesh Joshi

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one.

You're reading it!

Histogram of Oriented Gradients (HOG)

1. *Explain how (and identify where in your code) you extracted HOG features from the training images.*

I used the skimage hog function to extract the Histogram of Oriented Gradients. The whole pipeline for extracting the features can be seen in make_model.py from lines 62 to 87.

I decided to use the following features for hog features as they allowed a high level of accuracy out of the box:

```
# PARAMETERS
#=============================================================
orient = 8
pix_per_cell = 8
cell_per_block = 2
hog_depth = 0

useColor = True
useSpatial = True

spatial_size = (32, 32)
hist_bins = 5

colorSpace = 'RGB2YUV'
#=============================================================
```

2. Explain how you settled on your final choice of HOG parameters.

The parameters for the HOG features were taken from what was suggested in the lessons. With these hog features, my SVM managed to score an accuracy of 93%

To improve this, I decided to augment my feature vector with colour and spatial features.

Individually, the parameters below gave an accuracy of 90%

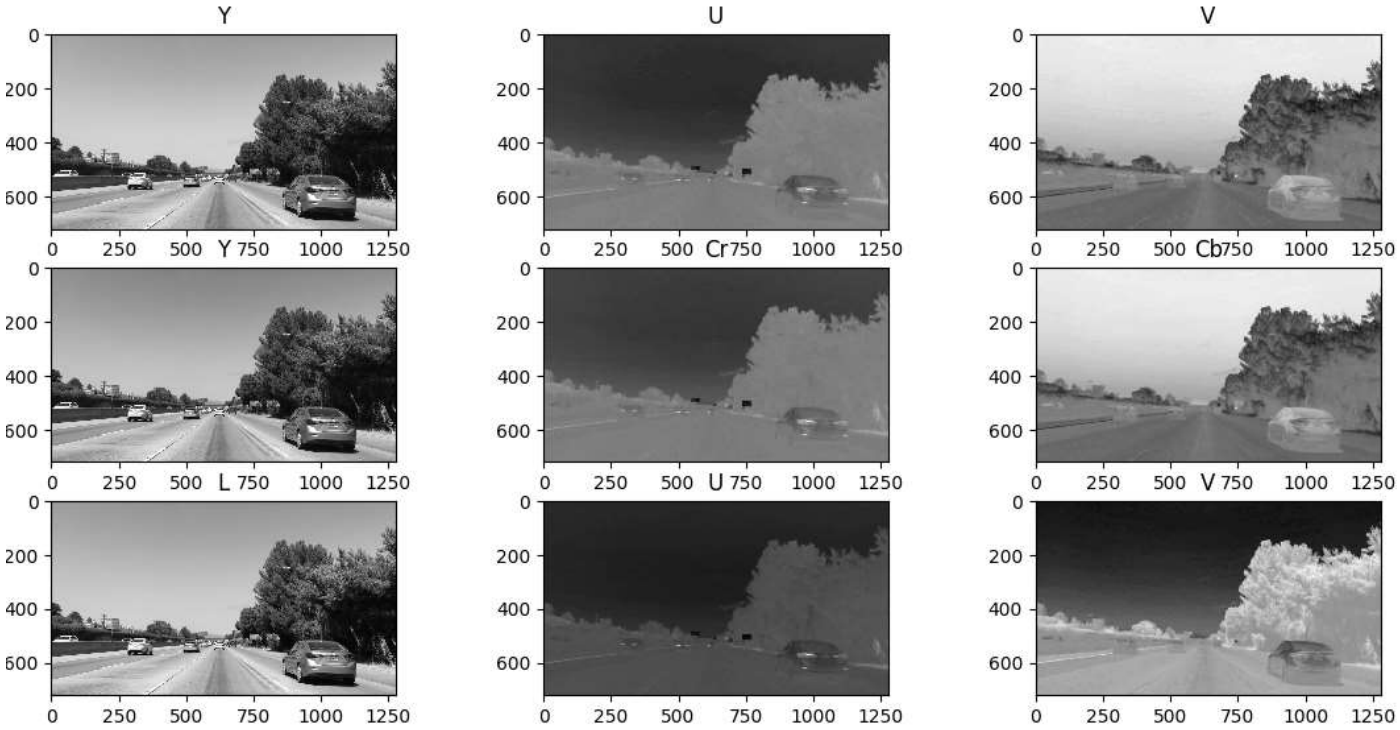
```
spatial_size = (32, 32) # Spatial binning dimensions

hist_bins = 5    # Number of histogram bins
```

Why hist_bins = 5

I decided to go with hist_bins = 5 because on inspecting the feature vector, I found that it was mostly sparse. I decided to use 5 bins as a denser representation

Why YUV colour space:

Initially, I had implemented a model with the RGB colour space. But that model was able to recognize the black car well but not the white car. To make a more robust colour system, I explored the colour spaces using `colorspace_explore.py` which showed the colour spaces.



I felt like the 0th channel had the right details for the task while the first and second lost most of it.
The Y channel of YUV somehow felt to capture some more contrast and I chose that.

After augmenting the HOG features with the spatial and colour features, I achieved an accuracy of 97% with a feature vector of size 4655.

97% test accuracy felt good enough for my application. So, I saved the classifier, scaler and the parameters as a pickle (`Models/ model_YUV0_hog_hist_spa.pkl`)

3. Describe how (and identify where in your code) you trained a classifier using your selected HOG features (and color features if you used them).

The code for this can be found in make_model.py from lines 90 to 107.
I started by shuffling the whole dataset so that the train test split don't have consecutive images.
Then, I scaled the data and split it into train and test.

The final accuracy was 97%

Sliding Window Search

1. Describe how (and identify where in your code) you implemented a sliding window search. How did you decide what scales to search and how much to overlap windows?

I implemented a sliding window search in `gen_vid_w_buffer.py` from lines 27 to 45

Before working on the video, the possible boxes to check are retrieved from `lesson_lib.py` using the `generate_search_windows` function, which calculates the boxes for the given size of the image.

Deciding scales and box sizes:
To get a feel of what my function was doing, I wrote two scripts:

1. **SubSampling Search**: This was implemented in `test_on_images.py` which computes the HOG features of the whole patch and then subsamples the image patches using the scale. However, this method handicapped me greatly in terms of overlapping windows and quality of results. After trying it a few times, with dissatisfactory results, I moved on to:

2. **Sliding Search**: This was implemented in `test_on_images2.py` which applies a normal sliding window. I tried different parameters until I settled on sampling:

	Image Size	Patch Overlap
1. Small Windows	**[128, 128]**	**[0.85, 0.85]**
2. Medium Windows	**[96, 96]**	**[0.85, 0.85]**
3. Large Windows	**[64, 64]**	**[0.85, 0.85]**

Note: This search was restricted to y: [390 to 720]

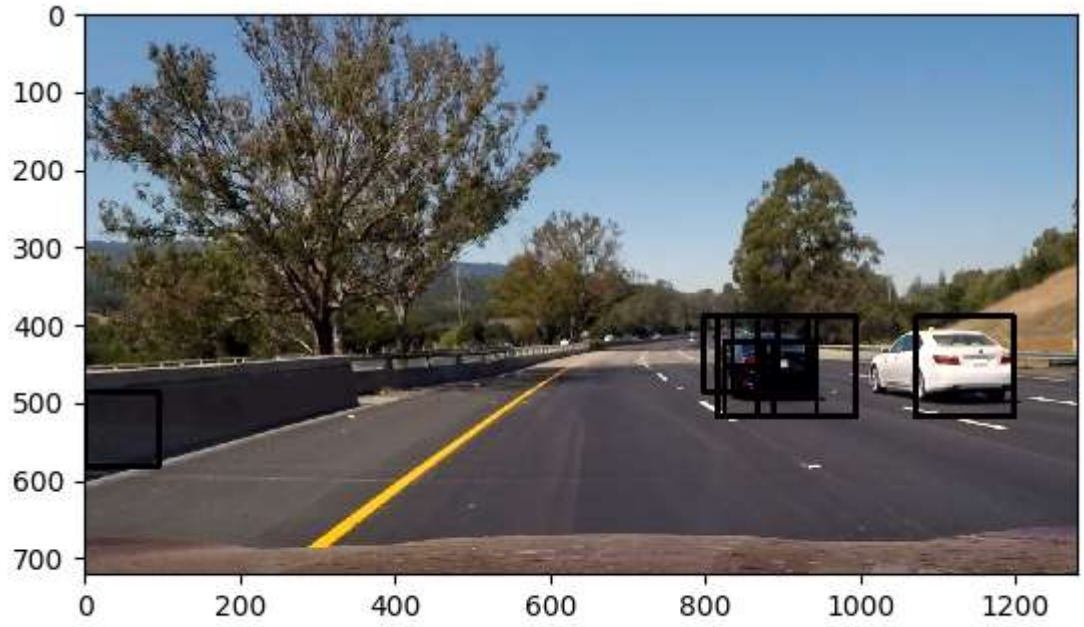
The parameter tuning was done using the images in `test_images/`

The second script gave the following results:

I unpickled my svm classifier, scaler and parameters which were trained on the provided data.

2. Show some examples of test images to demonstrate how your pipeline is working. What did you do to optimize the performance of your classifier?

The image below (& above) show how the svm classifier is able to catch some car patches as it slides over the image.



These hot windows are recorded and stored in a buffer which are used to remove false positives. Finally a heat map is generated of a certain number of frames and the prediction of car boxes is made.

To improve the accuracy, I didn't rely on a single window size. Instead, I used window sizes of 64, 96 and 128.

Search Window Optimization:
To prevent having to find the windows to search for each frame, I calculate them before the video loop and use the same set for each frame. This significantly brought down the time per frame.

Note: I also experimented with the hog image sub sampling, but it didn't work so well for me.

Video Implementation

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (somewhat wobbly or unstable bounding boxes are ok as long as you are identifying the vehicles most of the time with minimal false positives.)

The final video can be found in `output_video/FinalVideoWithBuffer.mp4` and will be linked in the ReadMe.

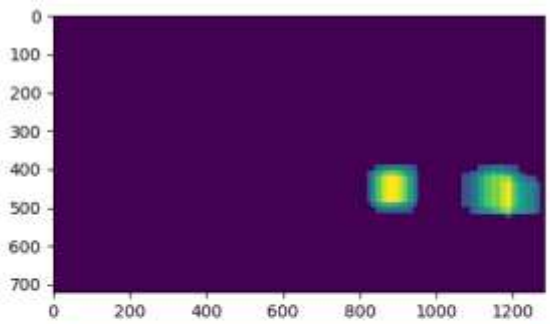
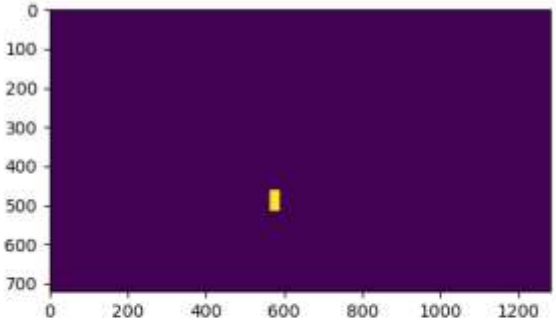
Note: I have also provided `output_video/FinalVideoWithoutUsingBuffer.mp4` to demonstrate the benefits of using a buffer.

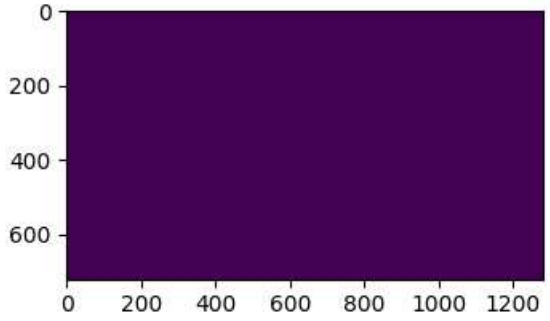
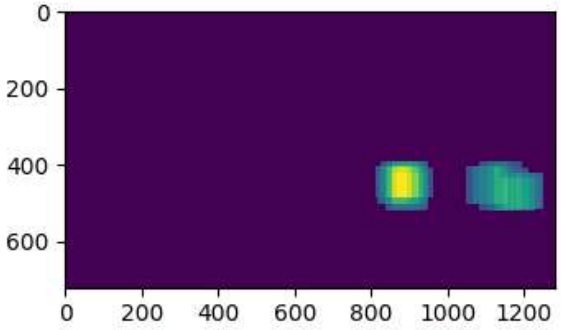
2. Describe how (and identify where in your code) you implemented some kind of filter for false positives and some method for combining overlapping bounding boxes.

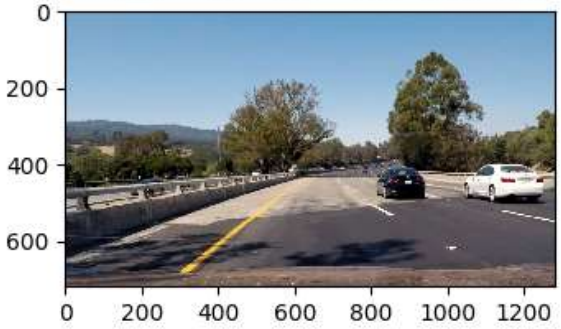
False Positives:
I eliminated many of the false positives by restricting the window that the search is done on to y = [390 to 720]

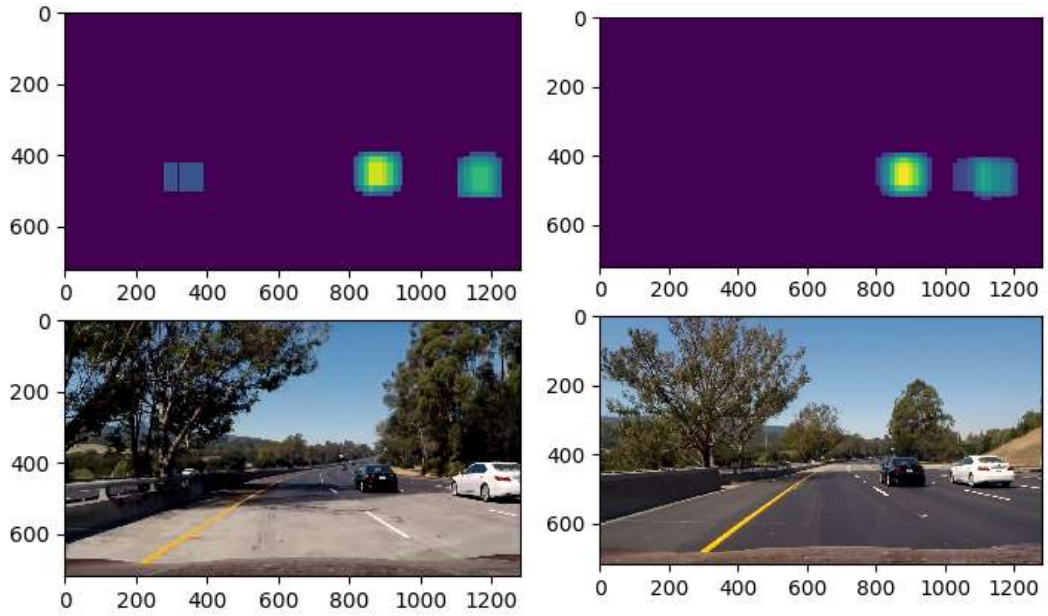
This still left a lot of false positives on the road which came up occasionally.
To overcome these sudden recognitions, I set up a buffer in gen_vid_w_buffer.py from lines 32 to 57. It checks the last 3 images and selects the heat map which has a heat of more than 12.

For Overlapping Bounding Boxes:
I used a heat map to combine overlapping boxes in `gen_vid_w_buffer.py` from lines 60 to 72





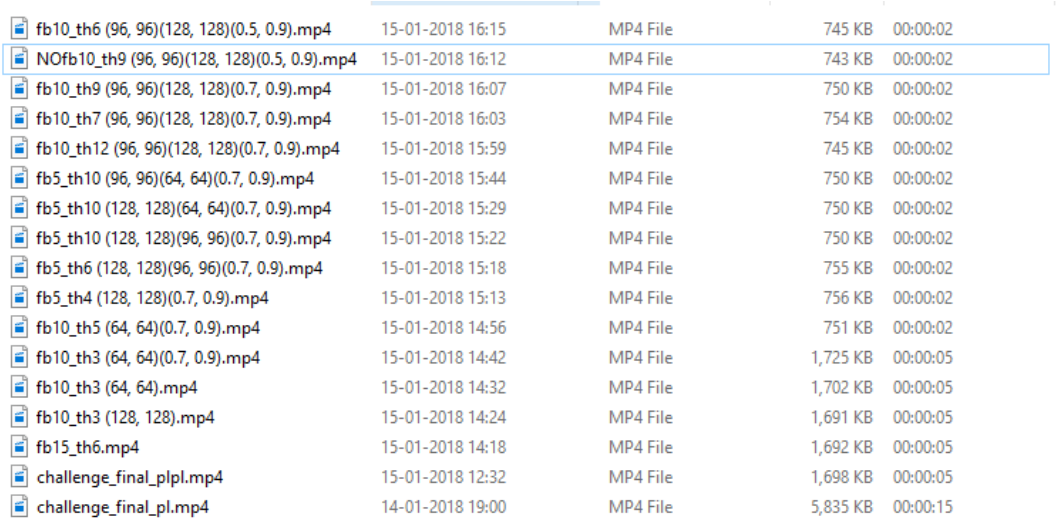
Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

My pipeline works well for cars but fails when it comes to false positives. This was overcome to a large extent due to my application of a buffer for the last 3 frames. However, false positives do occasionally pop up.
Moreover, my video loses track of the car at one point when the scale gets too small.

Problems I faced:
1. Time to generate videos:
There was a strong trade-off between a fast but less accurate method and a slow but accurate method. This was particularly worse for me as I trained on a CPU. Any prototype I generated took at least 15 minutes to test on a small clip of 15 seconds.
For generating the final full video, I had to keep the computer running overnight.
This could've been overcome with smaller window overlaps and hog image sub sampling but it didn't work so well for me.

2. Deciding Parameters:
This was another problem. It's difficult to get an intuition of what parameters will work. I ended up generating almost 30 videos at 15 minutes per video until I reached reasonable parameters.



fb10_th6 (96, 96)(128, 128)(0.5, 0.9).mp4	15-01-2018 16:15	MP4 File	745 KB	00:00:02
NOfb10_th9 (96, 96)(128, 128)(0.5, 0.9).mp4	15-01-2018 16:12	MP4 File	743 KB	00:00:02
fb10_th9 (96, 96)(128, 128)(0.7, 0.9).mp4	15-01-2018 16:07	MP4 File	750 KB	00:00:02
fb10_th7 (96, 96)(128, 128)(0.7, 0.9).mp4	15-01-2018 16:03	MP4 File	754 KB	00:00:02
fb10_th12 (96, 96)(128, 128)(0.7, 0.9).mp4	15-01-2018 15:59	MP4 File	745 KB	00:00:02
fb5_th10 (96, 96)(64, 64)(0.7, 0.9).mp4	15-01-2018 15:44	MP4 File	750 KB	00:00:02
fb5_th10 (128, 128)(64, 64)(0.7, 0.9).mp4	15-01-2018 15:29	MP4 File	750 KB	00:00:02
fb5_th10 (128, 128)(96, 96)(0.7, 0.9).mp4	15-01-2018 15:22	MP4 File	750 KB	00:00:02
fb5_th6 (128, 128)(96, 96)(0.7, 0.9).mp4	15-01-2018 15:18	MP4 File	755 KB	00:00:02
fb5_th4 (128, 128)(0.7, 0.9).mp4	15-01-2018 15:13	MP4 File	756 KB	00:00:02
fb10_th5 (64, 64)(0.7, 0.9).mp4	15-01-2018 14:56	MP4 File	751 KB	00:00:02
fb10_th3 (64, 64)(0.7, 0.9).mp4	15-01-2018 14:42	MP4 File	1,725 KB	00:00:05
fb10_th3 (64, 64).mp4	15-01-2018 14:32	MP4 File	1,702 KB	00:00:05
fb10_th3 (128, 128).mp4	15-01-2018 14:24	MP4 File	1,691 KB	00:00:05
fb15_th6.mp4	15-01-2018 14:18	MP4 File	1,692 KB	00:00:05
challenge_final_plpl.mp4	15-01-2018 12:32	MP4 File	1,698 KB	00:00:05
challenge_final_pl.mp4	14-01-2018 19:00	MP4 File	5,835 KB	00:00:15

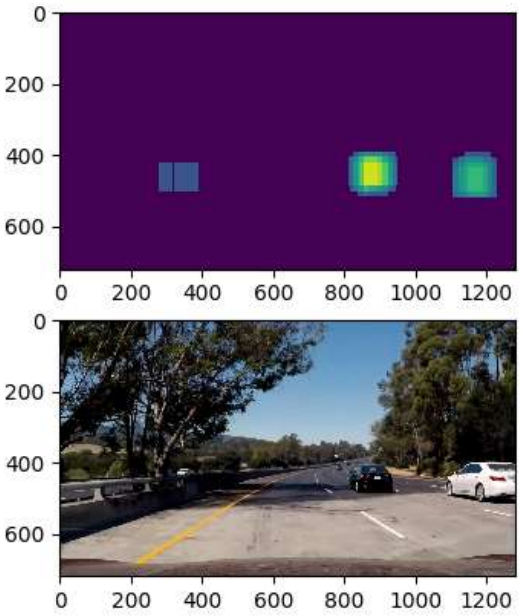
Where my pipeline might fail:

1. **Different Coloured Cars:**
   While colour plays a strong role in differentiating cars from non-cars, the training set might lead to some sort of bias towards certain cars.
   For example, in one of my prototypes with RGB colour features and hog features, my pipeline could recognize black cars perfectly but only managed to get a small section of the white car. Because of this, I later moved to the Y channel of YUV.

2. **False Positives on the road:**
   For some reason, my SVM chose the left lane line area as a car. This lead to too many false positives which occasionally flickered onto the screen. They could be mitigated with a frame buffer.

3. **Opposite Lane Cars:**
   As can be seen in the final project video, the model also considers images on the far left. It mostly classifies the cars in that lane which could distract our car. This can be mitigated by reducing the search width.

4. **Shadows and Noise:**
   As can be seen below, the svm gets fooled by shadow or noise patches which might look like cars.



What could you do to make it more robust?

Data Augmentation:

The model tends to create too many false positives on the road patches and in shadows. A data augmentation which artificially implants the non-car images with different darkness could potentially solve the problem.

Proper Parameter Search:

A full exploration of different parameters could possibly lead to a better solution. However, due to low compute resources, rapid prototyping has been almost impossible. The sheer time for processing has made this process stretch out too much. Surely, in my limited search, I might have missed out on the right set of parameters.