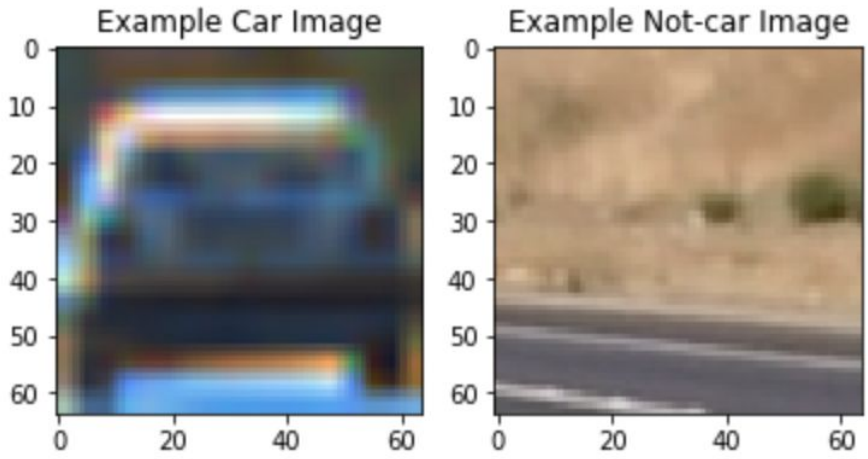
Vehicle Detection & Tracking

Prepare Dataset

The entire dataset contained **8792** car images and **8968** non-car ones, of size **(64, 64, 3)**, and data type **uint8**. This is an example of both classes:

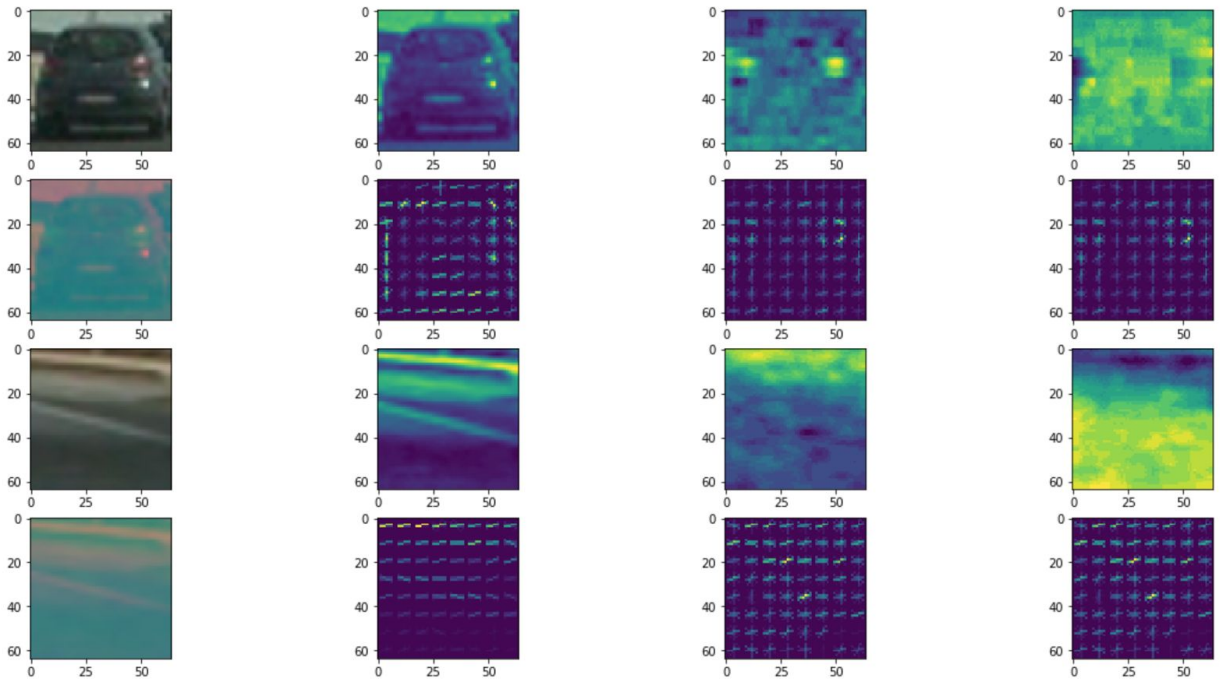


Feature Extraction

For better performance and speed, I decided to use as features just Histograms of Oriented Gradients for each of the 3 channels of the image, in the *YCrCb* colour space. I have come to this conclusion by trying out different combinations of features that included color histograms, raw pixel value bins, color channels and HOG. The gradients seemed to work the best in distinguishing between images with and without a car present. They are capable of better retaining the uniqueness of the data while also leaving space for variations that account for different representations of the same object.

After trying several values for each of the core parameters, I have extracted the features for each channel using skimage's **hog()** function with **11 orientation bins**, **8 pixels per cell**, and **2 cells per block**. For one image this gave a feature vector length of **6468.**

The following is an example of an extraction:



Data Processing

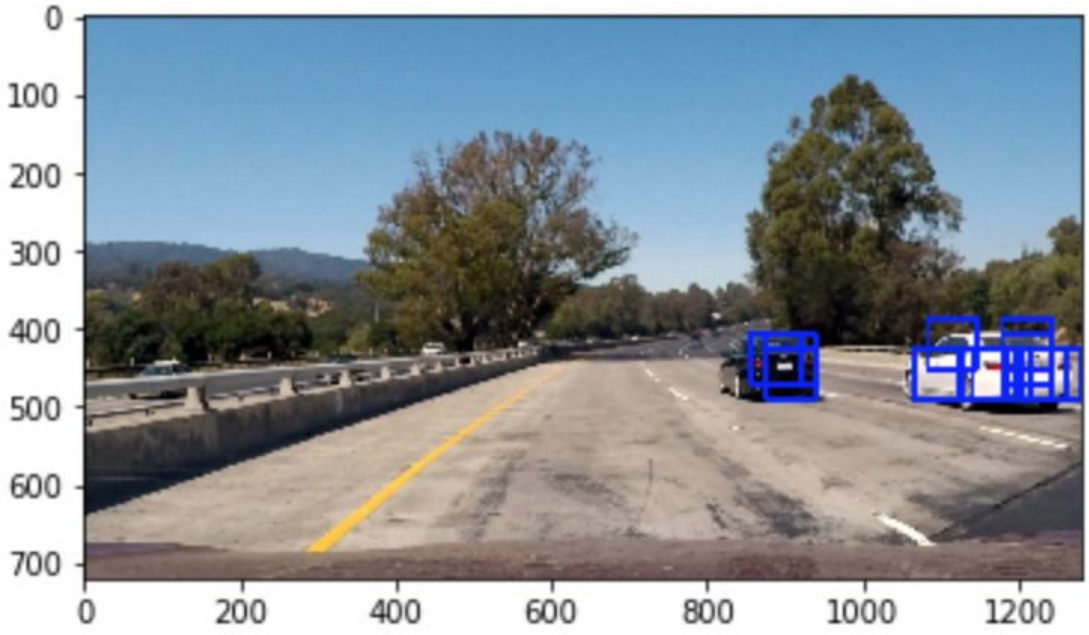
The extracted features set was normalized using the **StandardScaler** from scikit-learn, and then labels were created using Numpy's **ones()** function. Finally the resulting data set was shuffled and split into train and test sets with a 0.8 - 0.2 ratio.

Model Training

The model chosen for training was an LinearSVC, which is known to achieve good accuracy in a short period of time. The training of the model took ~16 seconds, and it scored an accuracy on the test data of **0.9882**.

Grid Search was also tried on the same model, but given the high accuracy achieved by the default parameters, I decided to stick with the basic configuration, which uses L2 as a penalization norm, calculates the loss using the square of the hinge loss, has a C penalty value of 1, and a tolerance for stopping of 0.0001. The model was trained for maximum 1000 iterations.

This is the result after testing the classifier on a test image:



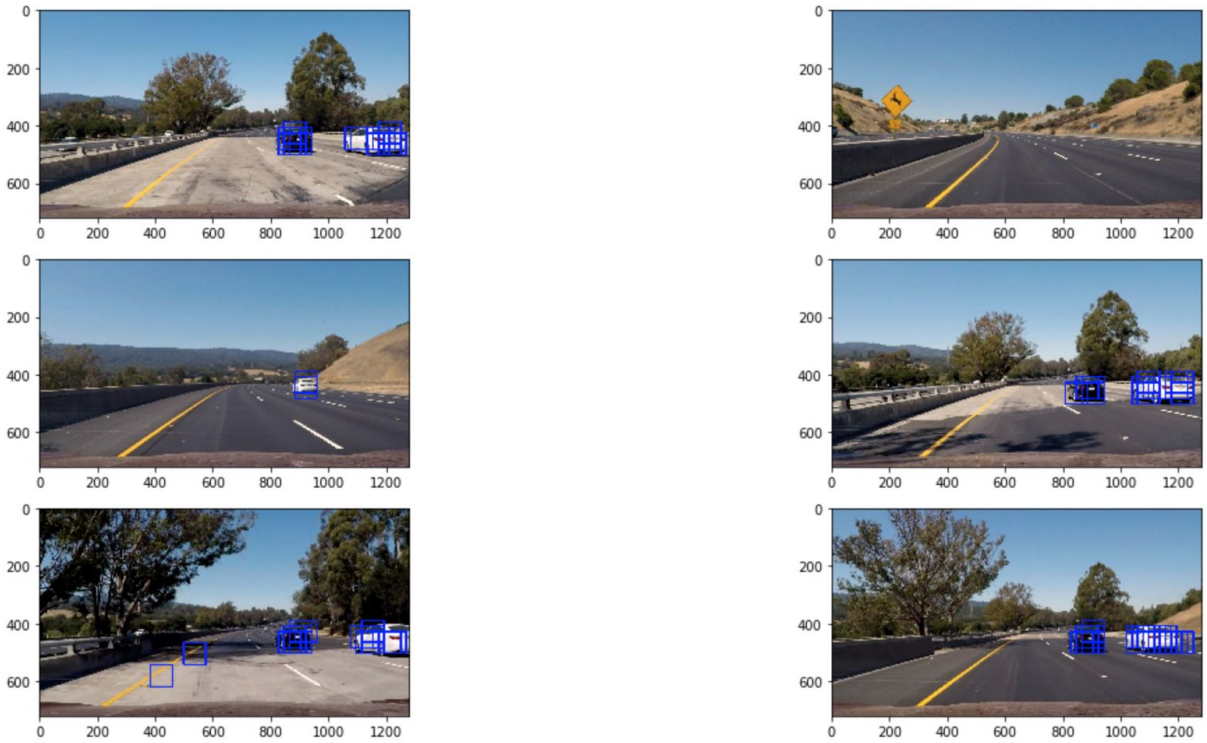
Sliding Window Search

I have started with the basic window search for extracting features of an image and I continued with defining several kernel dimensions to use. Although this gave good results - the above image was processed with this technique for prediction - I decided to use for the video the **HOG Sub-sampling Window Search** for speed reasons - instead of computing the HOG for each window, with this technique we can compute the HOG once for the entire image, and then use different kernels to sub-sample it for creating features and then predicting.

The HOG was taken over the entire X axis, and along the **Y** axis starting with **position 350** to **position 656**.

For scaling, values between 1 and 2 were tried. At the end **1.2** and **1.5** gave the most accurate predictions.

Here are some example images of the results on the test set:



Multiple Detections & False Positives

As it can be seen in the above images, multiple detections are visible through the overlapping bounding boxes, and also the two false positives from the 5h images are present.

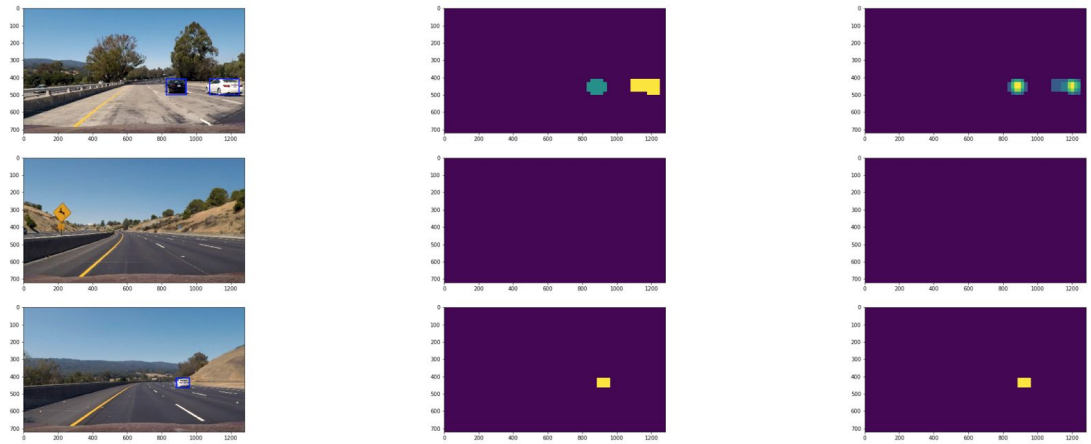
To solve for this problem in the video processing part, multiple consecutive frames were used to create a **heatmap**, and then **threshold** the map for filtering out the false predictions and multiple detections.

For each box detected, a 1 was added to an initially 0 values map with the same dimensions as the original image, over the area of the box. Hence places detected more than once will have larger values. By applying a threshold, we can discard all values that are lower than it, and hence keep just the predictions that were discovered over a sequence of images - these having a larger probability of detecting true cars. For distinguishing between prediction boxes in the thresholded map, *scipy.ndimage.measurements.label()* was used. Given that each

prediction box represent a detected vehicle, we can use the labels returned, to draw bounding boxes on the original image around the classified areas, using OpenCV.

These are the result for several of the testing images with a threshold of 1 - which means that all predictions identified only once in an area were discarded. Each row corresponds to the image and its heatmap:

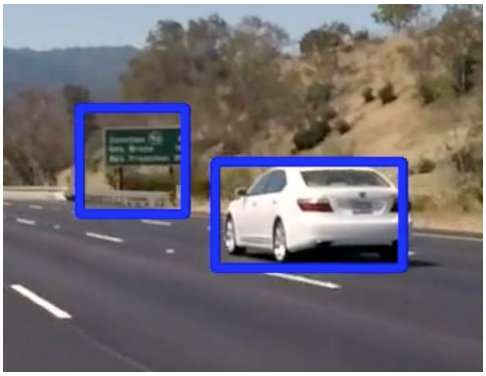
And the following examples include also their labels in the middle row:

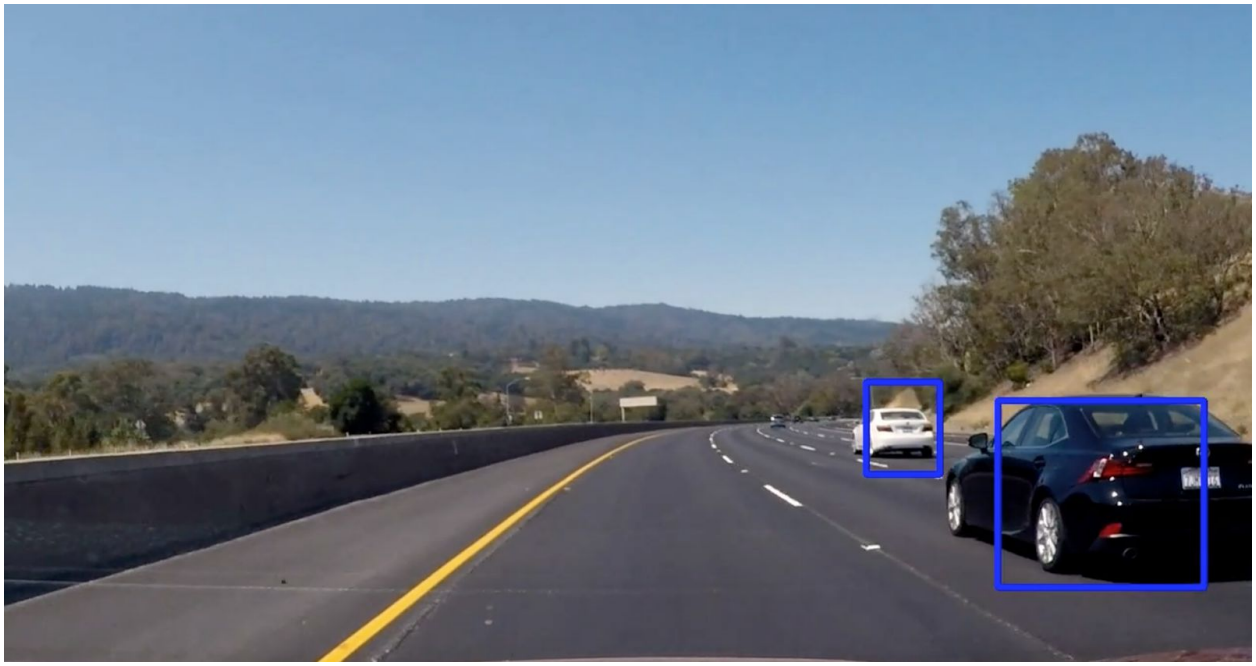


Vehicle Detection Pipeline

For processing the videos, I have used a **counter class** to keep track of the boxes over several frames - around **2** seemed to work the best, and to construct an aggregate heatmap from consecutive frames for eliminating noise and duplications. The **threshold value** used for the heatmap was **1**.

These are several examples from the final video:





Discussions

One of the challenges of this project was the elimination of outliers in the video frames. Even with the parameters tuned they still pop-up from time to time. The heatmap and its thresholding to a good job of lowering the false positives, but still some trade-off still exists between always recognizing all the vehicles, and eliminating all outliers. I believe the former is more important, because detecting a false vehicle will only constrain the car in making some actions, but failing to recognize a vehicle can end up with harsh consequences.

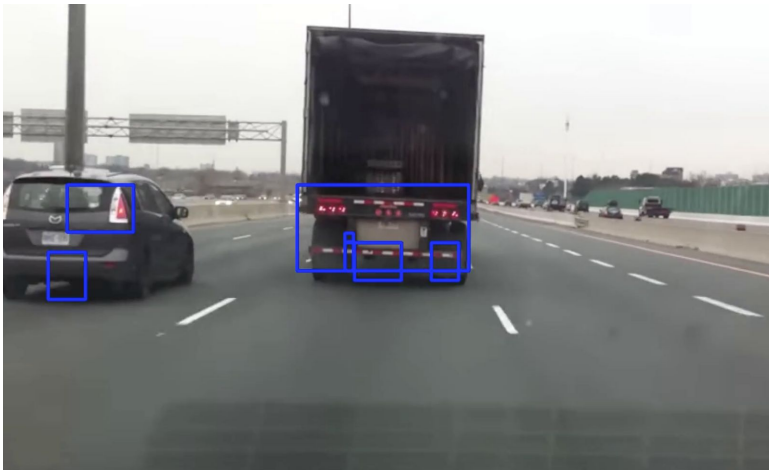
Another part of the artistry of this project is to create a pipeline that is highly accurate and low on processing needs. It seemed relatively easy to construct something that works, but developing a component that can run in real time, actually requires more in depth work, especially on trying different combinations of features and thresholds. Though really good at representing objects, HOG features seem to take longer to compute than the colour histograms. One solution for this would be to keep with just 1 channel for HOG, and add some colour histograms to help with the representation.

I have also tried the pipeline on a new recording on the highway, but the results were not so good as on the provided videos. The resolution of the video played an important part on the accuracy. When I ran the pipeline on an 420 resolution, NOTHING happened - no box was visible for any of the 3095 frames! Then, I decided to try it on the standard 1552 x 1000 resolution and the bounding boxes started to appear in large numbers in good and bad places,

but also the processing time was at least doubled. The scenery is also different from the provided videos - dozens of cars presented, driving on a highway entrance, a lot of signs present on the road at different times etc. So this could also be the reason for the low accuracy of the pipeline on it. Definitely another round of tuning of the pipeline would have given better results, and I plan to do it sometime in the next period. Also training the model with more complex data, like even sign roads, and other textures present in the real scenarios could help him generalize better.

In terms of other considerations for real scenarios, being able to distinguish between detected cars that drive in the opposite direction and cars that are on the other part of the road then ours (like on a highway) - and don't present a threat, seems to be really important.

Here are several takes from the 3rd video:




WEST / OUEST
Express
400 Barrie 1 km
Black Creek Dr. 1 km
Weston Road 3 km


Express