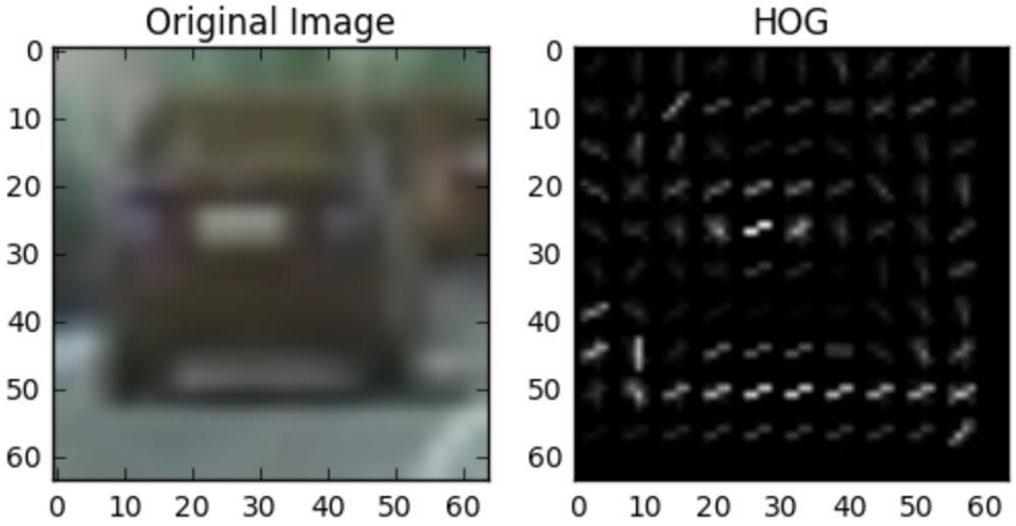
Vehicle Detection Project 5
By: Carlos Arreaza Feb 2017
Histogram of Oriented Gradients (HOG)

1) Explain how (and identify where in your code) you extracted HOG features from the training images. I was able to extract the HOG features from the training images in the function get_hog_features() which took as input the image itself, and some values like the orientation, the amount of cells inside of the blocks, amount of blocks per picture, etc. The function prototype looks like follows:

get_hog_features(img, orient, pix_per_cell, cell_per_block, vis=False, feature_vec=True), and returns '*features*', which is a vector containing the histogram of oriented gradients of the image 'img'. This function uses *hog* from *skimage.feature*:

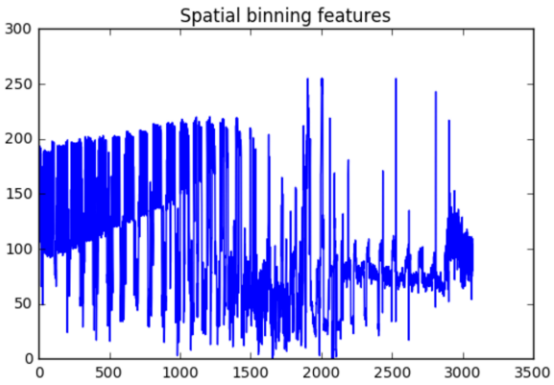
hog(img, orientations=orient, pixels_per_cell=(pix_per_cell, pix_per_cell),
cells_per_block=(cell_per_block, cell_per_block), transform_sqrt=False,
visualise=False, feature_vector=feature_vec)



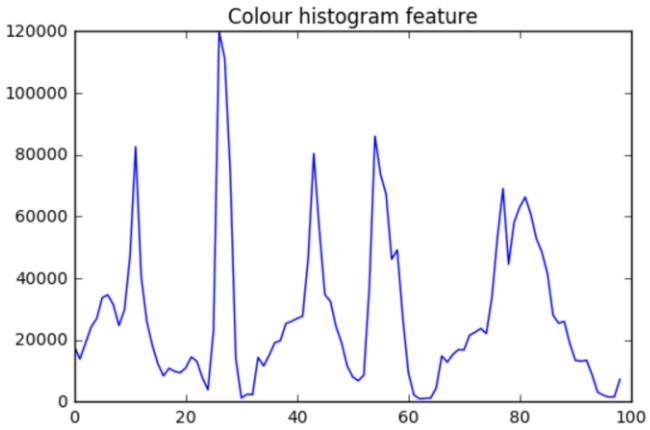
2) Explain how you settled on your final choice of HOG parameters.

Basically, I tried different numbers until I got the highest accuracy for the SVM classifier. I was able to get 99.3243243243% accuracy using the following parameters:

- Color space: LUV gave me the highest accuracy, RBG was also good, but gave me 98% accuracy on the test images.
- For the hog() function I used all the 3 color channels of the image, orient = 9, pix_per_cell = 8, and cell_per_block = 2
- I used spatial binning for part of the image features resizing the images to (16,16).



- I used color histogram, concatenating all three color channels (BGR), using 16 bins, and a range of (0,256).



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

All 3 image features (HOG, spatial bins, color histograms) are concatenated and extracted in *extract_features()* for the training images, and *in single_img_features()* for each video image. Finally, the features are normalized using the following from *sklearn.preprocessing*:

```
X = np.vstack((car_features,notcar_features)).astype(np.float64)
# Fit a per-column scaler
X_scaler = StandardScaler().fit(X)
# Apply the scaler to X
scaled_X = X_scaler.transform(X)
```

Then, a set of training images and test images are obtained using train_test_split() from sklearn.cross_validation using a randomized state:

X_train, X_test, y_train, y_test = train_test_split(scaled_X, y, test_size=0.2, random_state=rand_state)

Finally, the SVM classifier was trained using LinearSVC() from scikit learn:

```
# Use a linear SVC (support vector classifier)
svc = LinearSVC()
```

 # Train the SVC
svc.fit(X_train, y_train)

Finally, the accuracy was obtained using the following function:

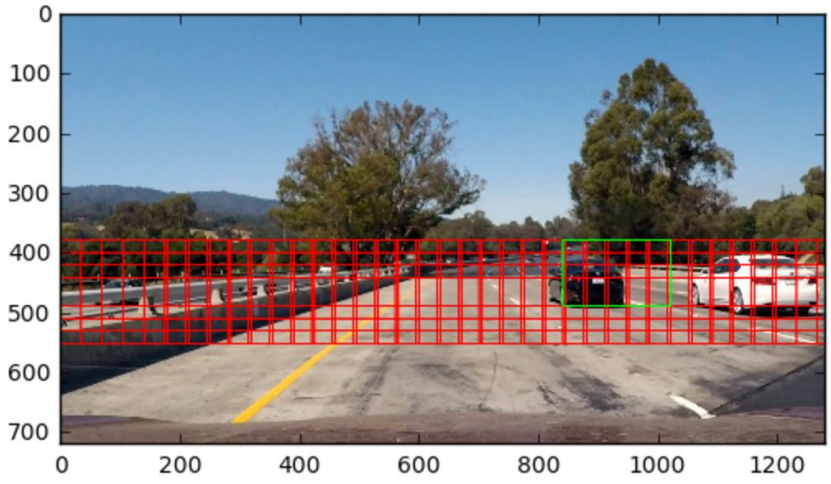
#Print accuracy
print('Test Accuracy of SVC = ', svc.score(X_test, y_test))

Sliding Window Search

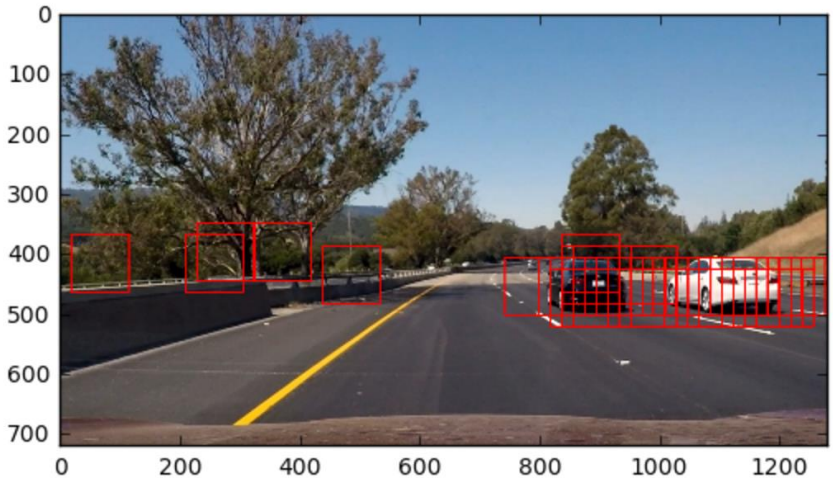
1) Describe how (and identify where in your code) you implemented a sliding window search. How did you decide what scales to search and how much to overlap windows?

The sliding window search was implemented in *slide_window().* This function outputs an array of 'windows' of basically 2 pairs of x,y coordinates. The scales of the windows were determined by eye. The closer the obstacles, the lower in the image and the larger the windows would be. The farther away the objects, the higher and smaller the images would be. I chose to use 3 scales: big, medium, small. Big (windows0 in the code): size (250, 160) with a y_start_stop=[400, 540] Medium (windows1 in the code): size (180, 110) with a y_start_stop=[380, 510] Small (windows2 in the code): size (120, 80) with a y_start_stop=[380, 450]. UPDATE 11-Feb-2017: I had way too many windows and thus was getting many false positives. I reduced the amount of windows, reduced the overlap to 0.8, and got much better results. I choose to limit the windows from y=350 all the way to 100 pixels above the bottom of the image.

windows = slide_window(image, x_start_stop=[None, None], y_start_stop=[350, image.shape[0]-100],
xy_window=(96, 96), xy_overlap=(0.80, 0.80))



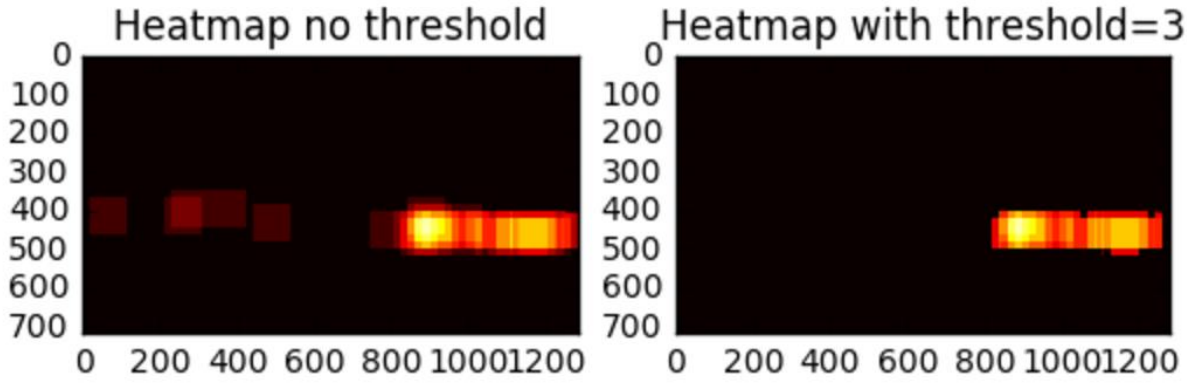
Using search_windows() inputting the already trained classifier we get the following for the test image:



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?

At the end of the notebook, I use my classifier and heatmap algorithm on all the test images. Improvements made to the algorithm:

- Added a heatmap to reduce the redundant boxes, choosing one box to surround the vehicle found. The heat map is created in function *add_heat(heatmap, boxlist).* The input of add_heat() are a numpy array of zeros, and the arrays of hot boxes (boxes were vehicles were found by the classifier). The output is the heatmap itself.
- Added a threshold of 3 to the heatmap to reduce the false positives. The threshold is called within function *heat_map_pipeline(image,windows)* and implemented in *apply_threshold(heatmap, threshold)*



- *Labels()* from scipy.ndimage.measurements is used to determine the amount of vehicles found fro the heatmap. Finally, we input the labels array to *draw_labeled_bboxes(img, labels)* to finally draw the boxes around the vehicles found.
- I was also able to reduce the noise of very small boxes using the line below. I decided to draw the boxes only if the size of the box is bigger than a specific amount (noise_red = 50):

  if ((np.abs(bbox[0][0]-bbox[1][0])>noise_red) & (np.abs(bbox[0][1]-bbox[1][1])>noise_red)) :

```
# Draw the box on the image
    cv2.rectangle(img, bbox[0], bbox[1], (0,0,255), 6)
```

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.)

Link to video: https://youtu.be/qJXyiaEASH0

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.

I was able to do this in my heatmap algorithm heat_map_pipeline(). The false positives were mitigated in apply_threshold() using a threshold of 4 to reduce the false positives. The overlapping boxes were added together, and if greater than the threshold, it was decided that it would be a selected vehicle. label() from scikit learn was used to determine the amount of vehicles found and separate them if multiple were found in the same image.

pipeline(image) is used to process images in the video. The output is the modified image with the boxes around the vehicles found.

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? Problems with this algorithm is the processing time, false positives, and if thresholds are too high it could create false negatives (when there is a vehicle present but it is not being found properly).

Pipeline could fail if vehicles were different from the ones shown in the dataset, or lighting conditions were different from that found in the dataset (night, snow, rain, etc.) To solve this problem one would have to find a bigger data set to better train the classifier. Maybe even use another type of classifier like a CNN.