Vehicle Detection Project

Summary

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
- Optionally, you can also apply a color transform and append binned color features, as well as histograms of color, to your HOG feature vector.
- Note: for those first two steps don't forget to normalize your features and randomize a selection for training and testing.
- Implement a sliding-window technique and use your trained classifier to search for vehicles in images.
- Run your pipeline on a video stream (start with the test_video.mp4 and later implement on full project_video.mp4) 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.

Rubric Points

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

1. Writeup / Readme

1.1 Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point

Your are reading it!

2. Histogram of Oriented Gradients (HOG)

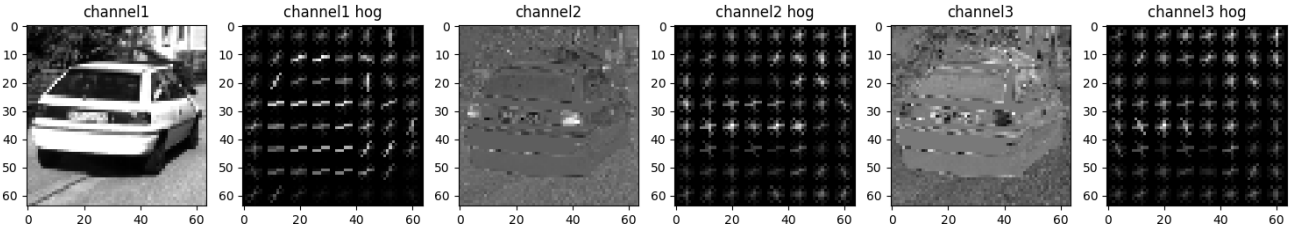
2.1 Explain how (and identify where in your code) you extracted HOG features from the training images.

The code for extracting HOG features from the training images can be found in "src/feature_extactor.py". In this module, I have created a class named FeatureExtractor, and all the functions responsible for feature extraction are placed here.

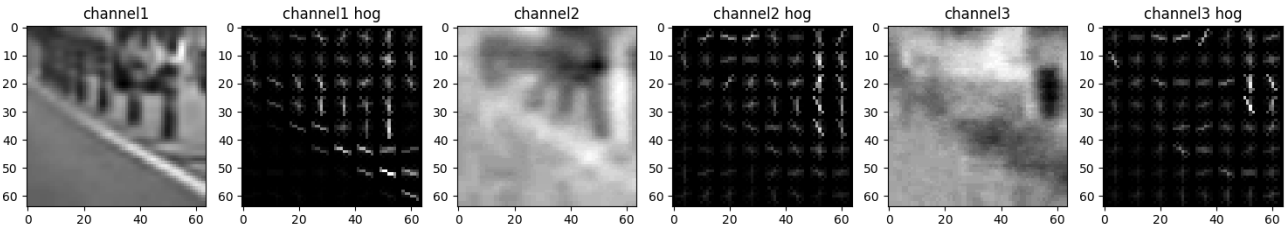
Static method `get_hog_features_from_pngs()` is the main function for extracting HOG features from given png file path list. For each of the image, it will convert it to the specified color space and use static method `extract_hog_features()` to extract the hog features from the image. In `extract_hog_features()`, it make use of `get_hog_features()` to do the real job by calling the `hog()` function from skimage.feature. For the details about those methods, please refer to the source code.

I have explored different color spaces as well as different parameters for the `skimage.hog()` including orientations, pixels_per_cell and cells_per_block. In the following, I showed some of my exploration of the method with the following randomly selected car and non-car images:

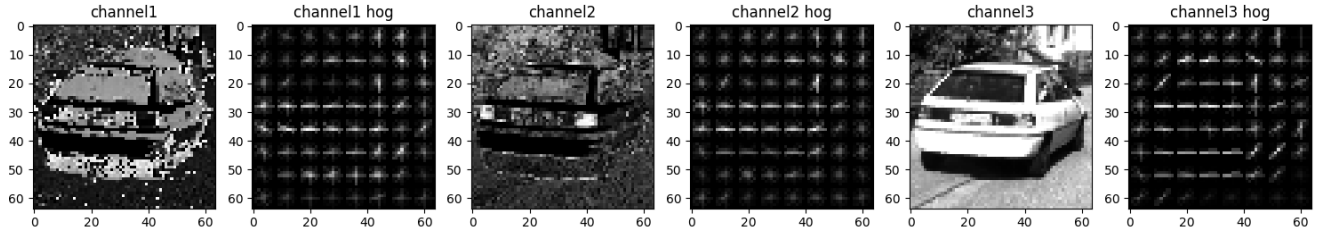
With color_space = "YCrCb", orientation = 9, pixels_per_cell=(8,8) and cells_per_block = (2, 2), I got the following hog features for each channel of the car image



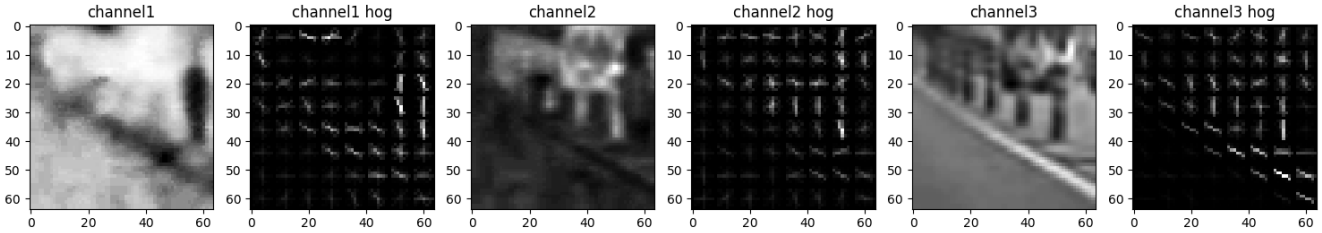
And for the non car image, I got the following hog features for each channel:



With color_space = ‘HSV’, orientation = 16, pixels_per_cell=(8,8) and cells_per_block = (2, 2), I got the following hog features for each channel of the car image:



And for the non car image, I got the following hog features for each channel:



2.2 Explain how you settled on your final choice of HOG parameters

Since it is difficult to select the HOG parameters visually, I selected my choice of HOG parameters based on their performance on classifying the cars. I have performed the following experiments by using only the HOG features of all channels, a linear SVC and the whole data set set from the course with 80~20 split for training and testing data. After those experiments, I have settled my final choice of the HOG parameters

1. Selection of color space:

With orientation = 9, pixels_per_cell=(8, 8), cells_per_block=(2,2), I tried different color space for the classification, and I got the following accuracy for different color space:

Color Space	Accuracy
YCrCb	0.9844
RGB	0.9656
HLS	0.9807
HSV	0.9787
YUV	0.9795

From this accuracy, it seems YCrCb performs the best, therefore, I selected YCrCb as my color spaces.

2. Selection of orientations:

After selecting the color space, in my experiment, I use YCrCb as the color space, and I fixed pixels_per_cell=(8, 8), cells_per_block=(2,2). By changing the orientations value, the classification accuracy results are shown as following:

Orientation	Accuracy
4	0.9670
6	0.9753
8	0.9812
9	0.9844
10	0.9858
11	0.9824
12	0.9849
13	0.9827
14	0.9829
15	0.9818
16	0.9827

From the table above, we can see that when orientation is >= 8, the accuracy is above 0.98, an even we increase the orientations, it does not change that much. However, with increase of orientations, the time used for extracting the features gets longer and longer. Finally, I selected 9 as the orientation.

3. Selection of Pixels per cell and cells per block

With similar method as above, after selecting the color space and orientations, I use YCrCb as the color space and orientation = 9. By changing the pixels per cell value and the cells per block value, the classification results are show as following:

Cells Per Block	Pixels Per Cell	Accuracy
(2, 2)	(4, 4)	0.9821
(2, 2)	(8, 8)	0.9844
(2, 2)	(16, 16)	0.9718
(4, 4)	(4, 4)	0.9792
(4, 4)	(8, 8)	0.9784
(4, 4)	(16, 16)	0.9724

From the table we can see that when cells_per_block = (2, 2) and pixels_per_cell = (8, 8), we get the best accuracy.

4. Final selection of HOG parameters

From the experiments mentioned above, I selected the following parameters for the HOG:

- color space: “YCrCb”
- orientation = 9
- pixels_per_cell = (8, 8)
- cells_per_block = (2, 2)

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

For the source code related to the classifier, please find source file “src/classifier.py”. In this module, I have created a class named Classifier. Its main job is to train, load and save car classifier.

1. Data Preparation

For the training data, I used the dataset provided in the course, it contains 8792 car images and 8968 non car images. However, to make the car set and non car set balanced, I used 8790 car image and 8790 non car images.

After extracting features from the image and before feeding the features for training, I used the `StandardScalar()` to remove the mean of features and scale them to unit variance. Then I randomly split the whole images to 80% for training and 20% for testing.

2. Features Selection

Besides the HOG features, I also considered the color features including the spatial features and the histogram features.

By using only the HOG features, I got accuracy 0.9844. By using HOG features with spatial features and histogram features, I got accuracy 0.9889. Therefore, In my final classifier, I used hog features together with color spatial features and histogram features.

3. Model Selection

For my classifier, I only used a LinearSVC, and from the result, it seems good enough. Therefore, I didn’t select other more complicate models. I think for more complicated models, it should also work, but it will take long time to train or predicate.

3. Sliding Window Search

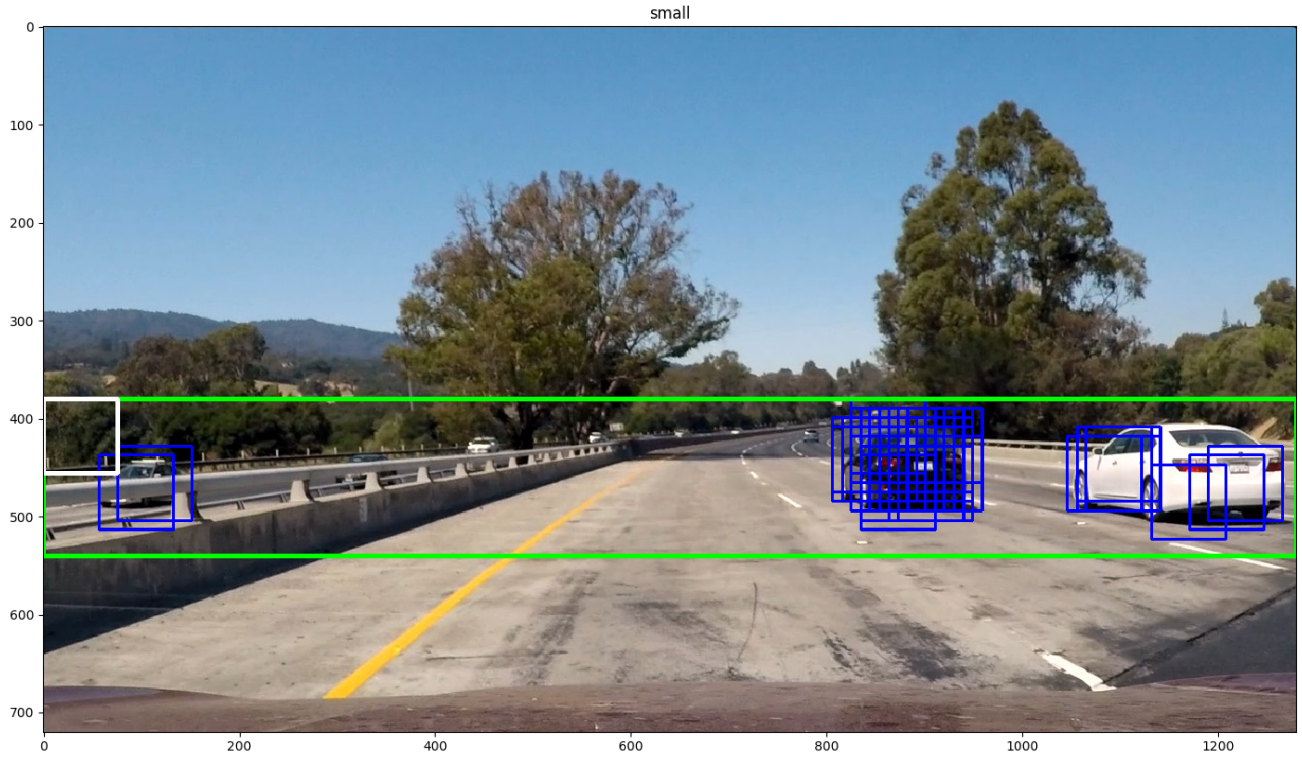
3.1 Describe how (and identify where in your code) you implemented a sliding window search. How did you decide what scales to search and how much to overlap windows?

I used the Hog sub-sampling window search method mentioned in the course for sliding window search. The code can be found in “src/pipeline.py”. In this module, I created a class called Pipeline, which is responsible for finding all cars in a given image and output an image with bounding boxes. The method find_cars() is doing sliding window search in an image, and method scan_all_cars() is to perform the sliding window in different scales, regions to try to find cars of different size in an image. The detail of the implementation can be found in the source code.

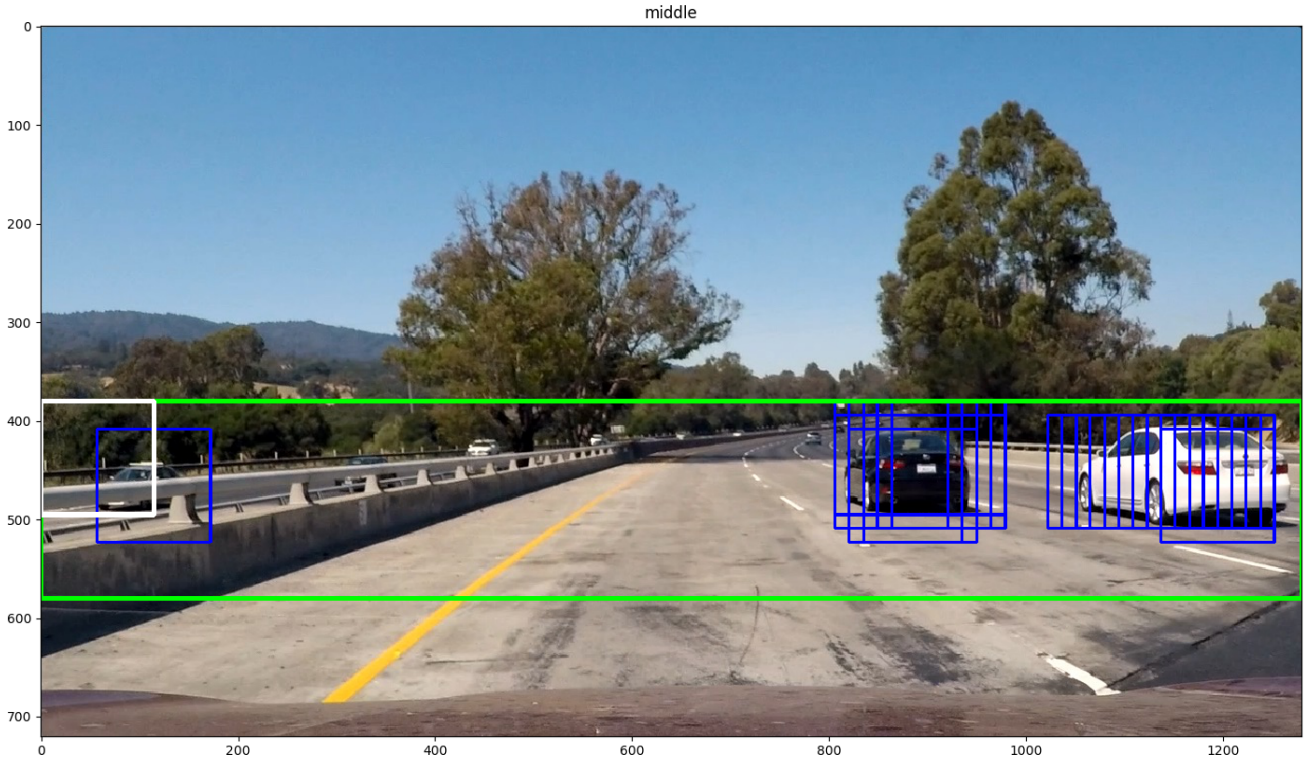
1. Different Scales

In my implementation, I used three different scales to search in the image. In the following images, the green rectangle shows the region of interest for this sliding window, and the white square shows the size of the sliding window. The blue ones shows the detected bounding boxes for cars.

The first scale is 1.2. It is responsible for small cars in the image. With the interested region of start y = 380 and end y = 540. It is shown as following:



The second scale is 1.6, it is responsible for middle size cars in the images. Its interested region has start y as 380 and end y as 580. It is shown as following:



And the last scale is 2.4, it is responsible for big cars in the image. Its interested region has start y as 380 and end y as 640. It is shown as following:

2. How much to overlap windows

Since I am using Hog sub-sampling window search method, the overlapping window is measured by the number of cells of moving the sliding window. In the original implementation of the sub-sampling window search method, it is using 2 cells in both x and y direction moving of the sliding window.

However, I found in y direction 2 cells is normally too big. Therefore I changed it to 1. Also in x direction 2 cells can also be too big some times. I changed the x direction move step to be related to last detection. If the last detection found a car, I will use 1 for the next slide along x direction, if the last detection did not find a car, I will use 2 cells for the next slide along x direction. For the details, please find in code "src/pipeline.py" class Pipeline method `find_cars()`.

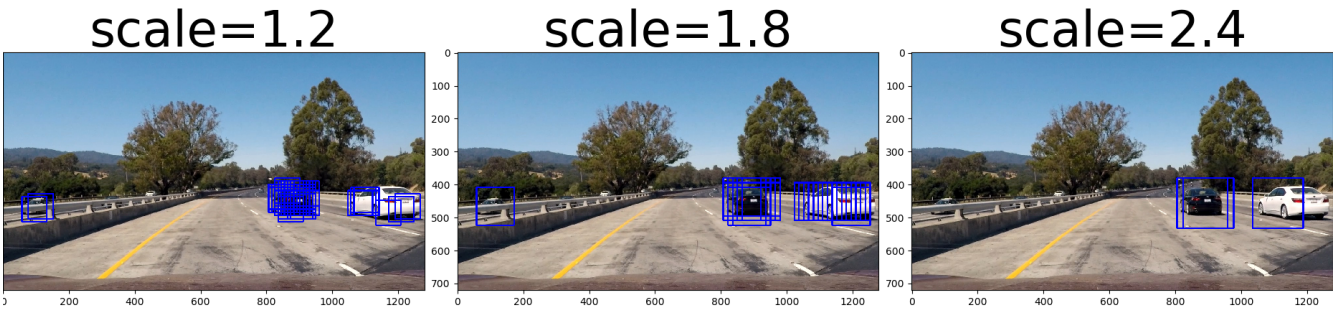
Here the compromise has to be made between the processing time and the detection result. The smaller step along x and y direction, the longer time it will take to process the image, but the better result will be achieved.

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

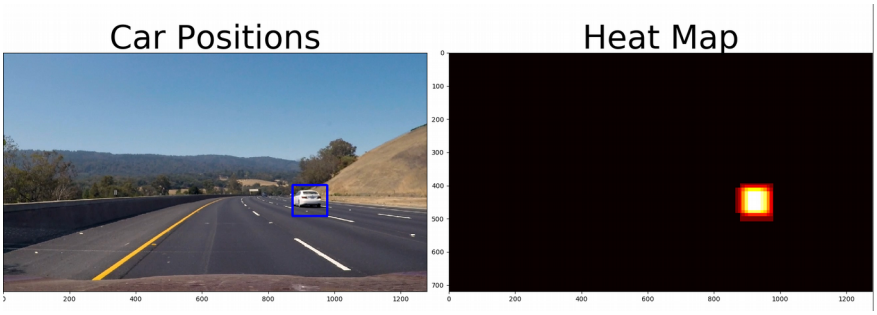
In my pipeline, I made use of hog features, color spatial features and color histogram features as the feature vector. The hog features is obtained as mentioned in above sections.

As shown in the last section, I first use the sliding window search to search the image for cars in 3 different scales and different region of interest. And Then I combine all the detected bounding boxes to get a heat map. And then, I applied a threshold to the heat map to remove those false positive detections. And finally, use the label() function on the filtered heat map to generate the final bounding box of the detected cars in the image.

The followings are some examples of applying the pipeline to the test images, from the result, it shows the pipeline works quite well.



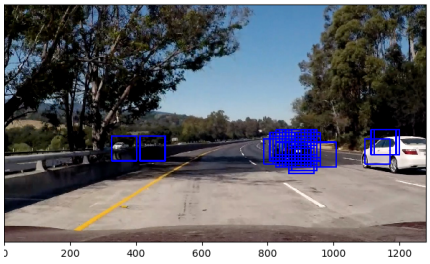

scale=1.2
scale=1.8
scale=2.4


Car Positions
Heat Map


scale=1.2
scale=1.8
scale=2.4


Car Positions
Heat Map

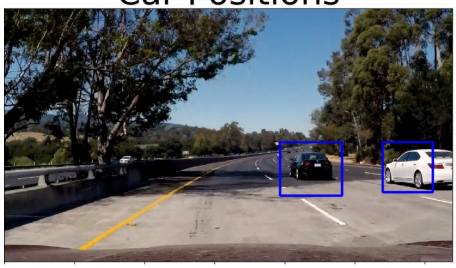
scale=1.2



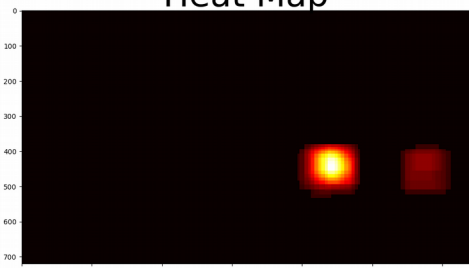
scale=1.8

scale=2.4

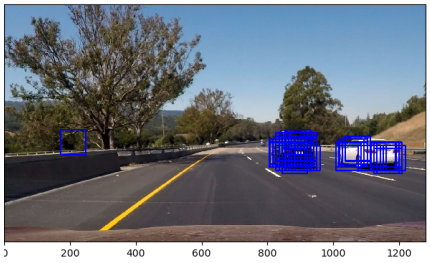
Car Positions



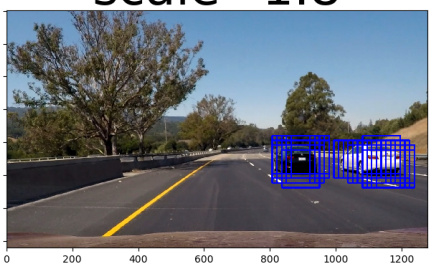
Heat Map



scale=1.2



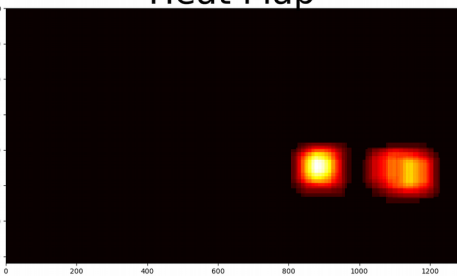
scale=1.8



scale=2.4

Car Positions

Heat Map



4. Video Implementation

4.1 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 link to the video result is as following:

https://www.youtube.com/watch?v=ADCGWk6uhfo

4.2. 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.

As mentioned in section 3, in my pipeline, I first use the sliding window search to search the image for cars in 3 different scales and different region of interest. And Then I combine all the detected bounding boxes to get a heat map. And then, I applied a threshold to the heat map to remove those false positive detections. And finally, use the label() function on the filtered heat map to generate the final bounding box of the detected cars in the image.

Besides, to make the result stable and smooth, I output the detection result by averaging over the last 10 frames' heat map and apply another threshold to the result. In this way, I can further remove the false positives and also the jittering.

5. Discussion

5.1. 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?

1. Problems Faced in the implementation:

Since we are using the different scales of sliding window to detect car in the image, for a single image, there could be many times of testing. To achieve good detection result, we need to use many different scales of sliding window and use small step to slide. However, this will increase the processing time. So the compromise between detection result and processing time is an issue has to be solved.

2. Cases where the pipeline is likely to fail:

Of course, under extreme lighting conditions, the car classification may fail.

Also in my pipeline, I used only scale of value 1.2, 1.8 and 2.4, for any car size in the image smaller than the size for scale 1.2 or bigger than the size for scale 2.4, my pipeline can not detect.

3. Measures to make the pipeline more robust

In my implementation, I am only using the vehicle and non vehicle coming with the project. Though the SVC already has a quite high accuracy, however, it still has some false positives from time to time. To further improve the robustness, one method is to use more training data. As introduced in the project, the dataset from Udacity can be a good source of additional data for the car classification.

To improve the classifier, another way would be trying different models like Decision Trees, Neural Networks, this may also improve the accuracy of classification.