Vehicle Detection and Tracking

Report on Udacity CarND Project No. 5, Nov. cohort 2016

Albert Killer, Mar 2017

1 Project assignments

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.

The above mentioned steps are implemented within following files, all in all containing around 1,000 lines of Python code:

```
tracking_pipeline.py
my_lesson_functions.py
lane_finding.py
```

The following approach is inspired by Udacity [1] and code provided by Udacity within the lessons was used and modified for this project’s purpose.

2 Histogram of Oriented Gradients (HOG)

2.1 Feature Extraction

Different ways are known to cope with the computer vision task of identifying objects, like vehicles or pedestrians on digital images. Those images can be analyzed regarding to their color and shape information for instance, to search for certain features. In order to get results which are reliable and more independent to changes in appearance (i.e. view from different angle or distance) we can take advantage of color value histograms. The function `color_hist()` used within the submitted code combines the histogram of each color channel to a single feature vector. In addition we convert the images from RGB to YCrCb color space to achieve color invariance.

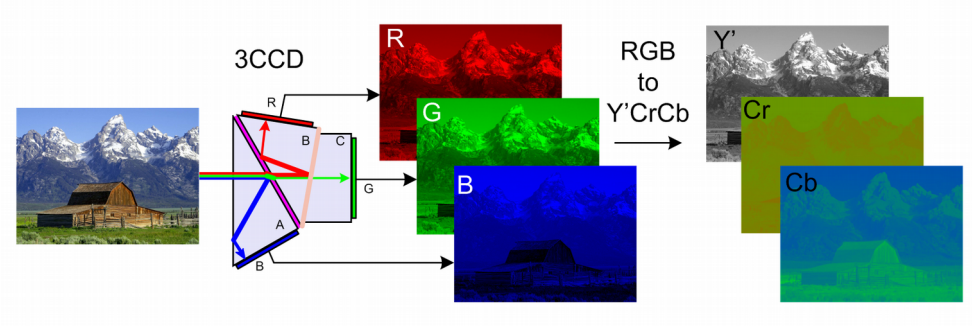
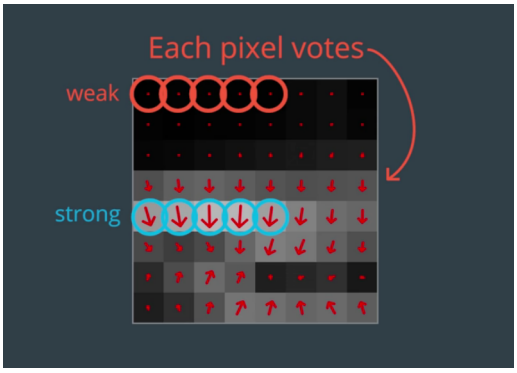


Figure 1: RGB to YCrCb conversion [3]



Another very helpful tool and widely used feature descriptors is the histogram of oriented gradients (HOG), which can be implemented using the Python library `scikit-image` [2]. HOG features describe gradient directions along a predefined pattern (of cells), giving each pixel in an image a vote on the overall gradient direction of its cell (see Figure 2, on the right [1]). "Stronger gradients contribute more weight […], and the effect of small random gradients due to noise etc. is reduced." [1] All in all making the HOG feature extraction a method robust to variation of shapes and at the same time giving a lot of room for fine tuning due to various parameters. Therefore we define the function `get_hog_features()` and set the number of *pixel per cell* and *cells per block* as well as the "number of *orientation bins* that the gradient information will be split up into in the histogram" [1]. And finally, when used for HOG feature extraction, the second color channel of YCrCb color space turned out to bring the best results in the present case (for a comparison of RGB, HSV and YCbCr, see Figure 3 below).
Following code snippet gives an overview of the main parameters and their values set accordingly, which can be tweaked to optimize feature extraction.

```
color_space = 'YCrCb'    # Can be RGB, HSV, LUV, HLS, YUV, YCrCb
orient = 9               # HOG orientations, usually between 6 and 12
pix_per_cell = 14        # HOG pixels per cell
cell_per_block = 2       # HOG cells per block
hog_channel = 1          # Can be 0, 1, 2, or "ALL"
spatial_size = (32, 32)  # Spatial binning dimensions
hist_bins = 196          # Number of histogram bins
```

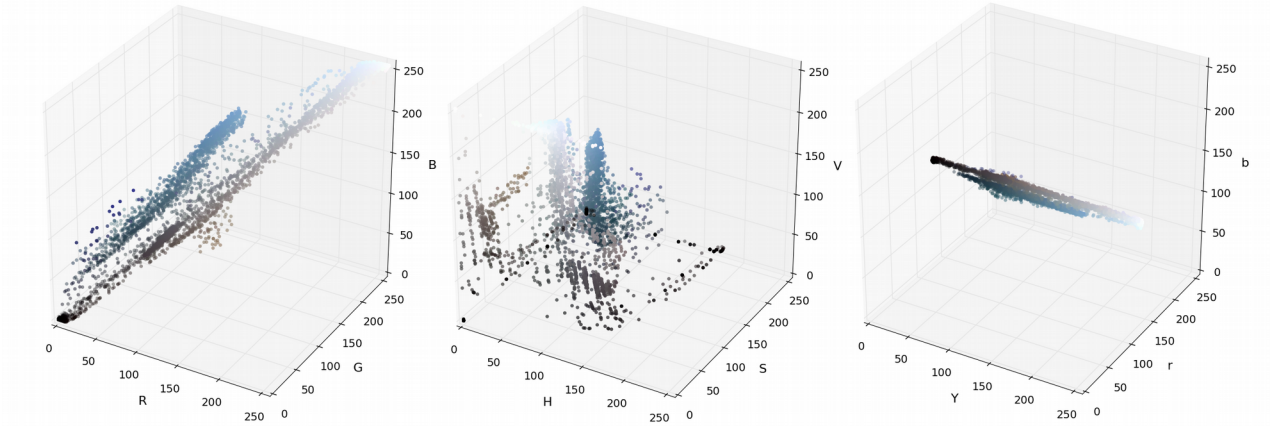


Figure 3: RGB, HSV and YCrCb color spaces (from left to right) of a 64x64 sample image.

2.2 Training the Classifier

The main goal of this project is to apply supervised learning by training a linear classifier to successfully detect vehicles within images captured on a highway. For this training we need a fair amount of samples with corresponding labels. We can generate the training data by importing a subset of an image collection provided by Udacity for this purpose, all in all including approx. 8,800 vehicle and non-vehicle samples from *GTI vehicle image database* [4] and the *KITTI vision benchmark suite* [5]. Then the first step is to apply the functions described in the previous chapter in order to extract relevant features (i.e. HOG feature extraction, steps visualized in Figure 4, below).

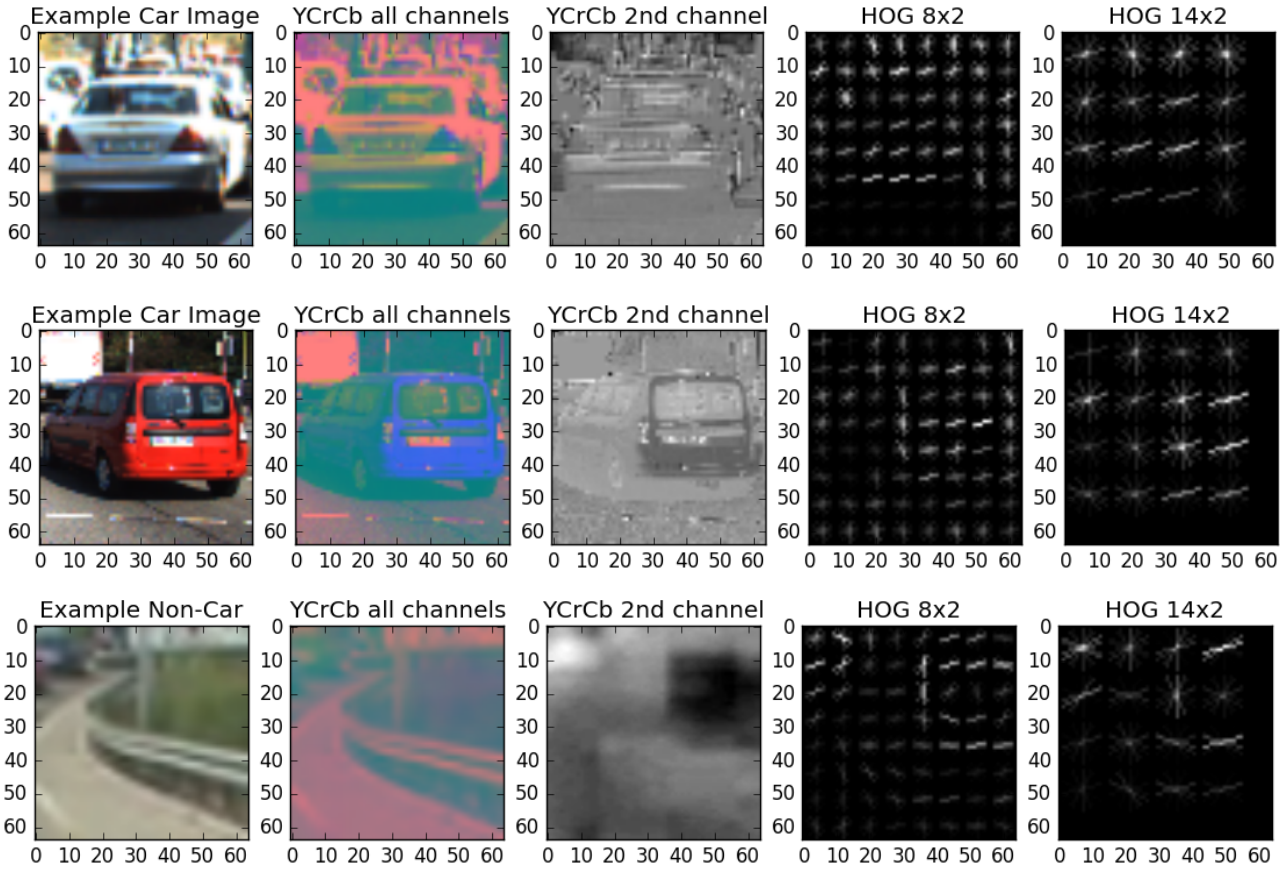


Figure 4: Stages of HOG feature extraction from YCrCb sample images.

During previous steps we have combined different features based on color histogram, color space and HOG in order to get all the relevant information extracted. As a result of combining those different methods, different quantities are represented within those features, which is why we want to normalize our data using `StandardScaler()` provided by Python's `sklearn` package. Before we start to train the classifier we split the images into training (80 %) and test set (20 %) and shuffle the data to avoid any potential side-effects caused by ordering.

For this task we use a Support Vector Machine (SVM) classifier and pass training set `X_train` with resulting feature vectors of length 3,984 each. The following code snippet shows how to create and train this linear classifier type using only two lines of code after importing `LinvearSVC` from `sklearn.svm` [6]:

```
svc = LinearSVC()
svc.fit(X_train, y_train)
```

3 Sliding Window Search

Having the classifier trained successfully with estimated accuracies above 99 %, we can now proceed to the actual pipeline for vehicle tracking. The designed algorithm provides an option to switch between an input mode for testing single images imported from local folder `test_images` and a mode for reading in frames of a video file, which is described in the next chapter. Both operational modes use the `slide_window()` function to scan the image by moving square sized "windows" across a certain area in steps defined by *window size* and *overlapping ratio (see Figure 5, below)*. On basis of these two parameters we calculate *number of pixels per step*, loop through each window position and finally return a list of all the windows. In the next step the list is handed over to the function `search_window()` where our methods of feature extraction are applied to each window. The extracted features are fed to the trained classifier to make its prediction on presence of potential vehicles. Windows with positive detections are returned as `hot_windows`.

Figure 5: Visualization of an area searched by "sliding windows".

The approach was taken to only search in one scale but therefore optimize the window's size and overlapping to the extend that all cars in the relevant field of view can be detected reliably. Figure 6 below shows an increase of windows size and overlapping resulting in different outputs on the test images: bounding boxes get less tight (subject to discussion which is preferred here), smaller vehicles are detected faster but at the same time false positives are likely to appear more frequently.

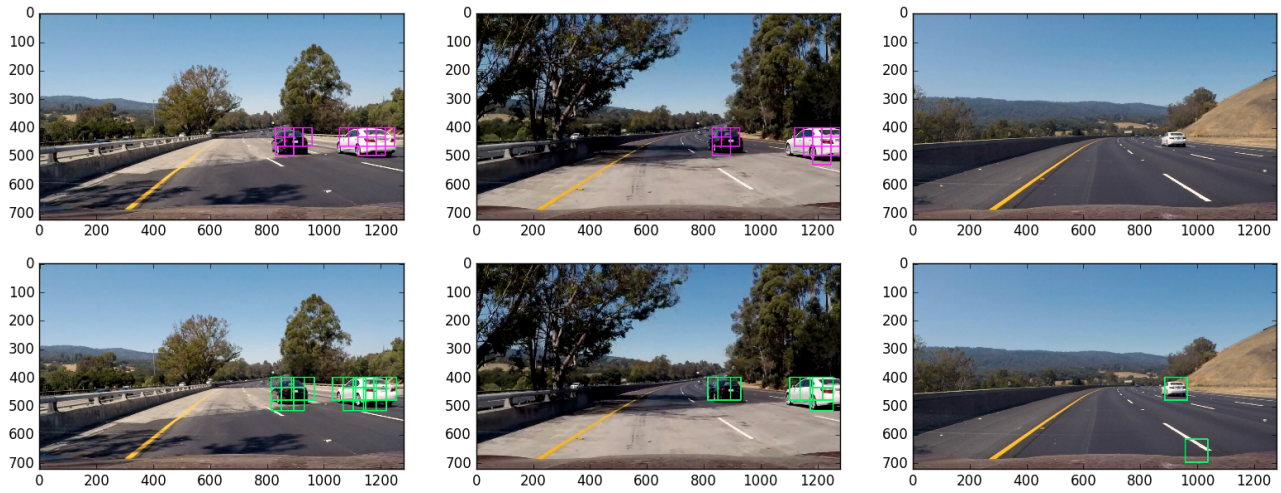


Figure 6: Window size/overlapping ratio 64x64/0.50 (above) vs. 80x80/0.55 (below).

In order to improve reliability by reducing false positives and multiple detections of vehicles, we can experiment with a range of various techniques. One of the most promising approaches described in the Udacity lesson [1] starts with creating a heat map which carves out the location of repeated detections by iterating through the list of `hot_windows` and stepwise increasing the value of every pixel within each window. As as result the function `add_heat()` returns an image with highlighted overlapping detections (cp. Figure 7, below). Afterwards all pixels below a threshold of 0.75 are zeroed out. To finally get the corrected location of our detected vehicles we use the `label()` function from `scipy.ndimage.measurements` package.
Furthermore the upper half of the image, mainly containing sky and trees, as well as the far left (opposite lane) was excluded from window search to further reduce unnecessary false positives, hence ignoring areas which are not relevant for the operation of a self-driving car (visualized in Figure 5, above).

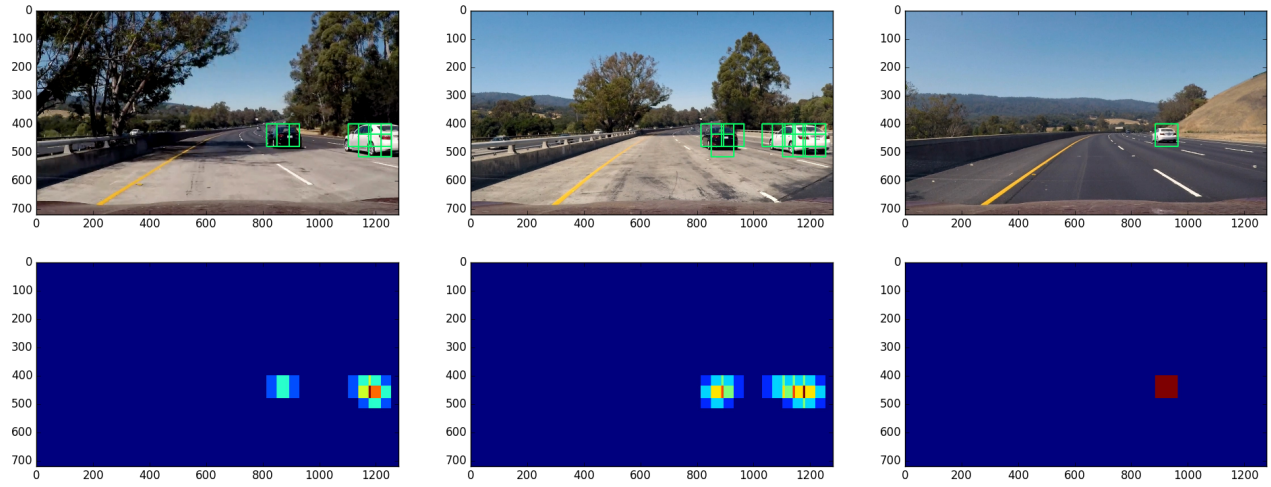


Figure 7: Comparison of heat map outputs.

4 Video Implementation

It is useful to apply the detection methods on single images for testing, especially to be to able to compare different outputs during parameter tuning. However the final goal is to implement a pipeline for processing videos and generating an output of the same video augmented by visualized vehicle detections. For this purpose we create the function `process_video()` which reads in a video file frame by frame and applies the same functions described previously in the chapters above.
To further enhance verification using the heat map method we can collect a certain amount of subsequent frames from the video and compare them to each other using a logical AND function. In doing so we can confirm new detections based on the fact that positive detections are likely to be found nearby previous ones. Best results are achieved by taking 7 or 8 frames into account, creating a smoother visualization of output, in other words more stable bounding boxes, and at the same time reducing false positives to nearly zero. As a downside of this method we accept that new vehicles are detected with a slight delay.

The final result shows a reliable vehicle detection in the majority of output frames, even though wobbling bounding boxes and occasional false positives are subject to further optimizations (cp. Chapter 5). In addition the vehicle detection pipeline is combined with the lane finding implementation of the previous project no. 4 “Advanced Lane Finding”. This way the output is extended to a visualization of the estimated lane as well as displaying values for measured lane line curvature and distance of the vehicle’s center to the approx. center of the lane.

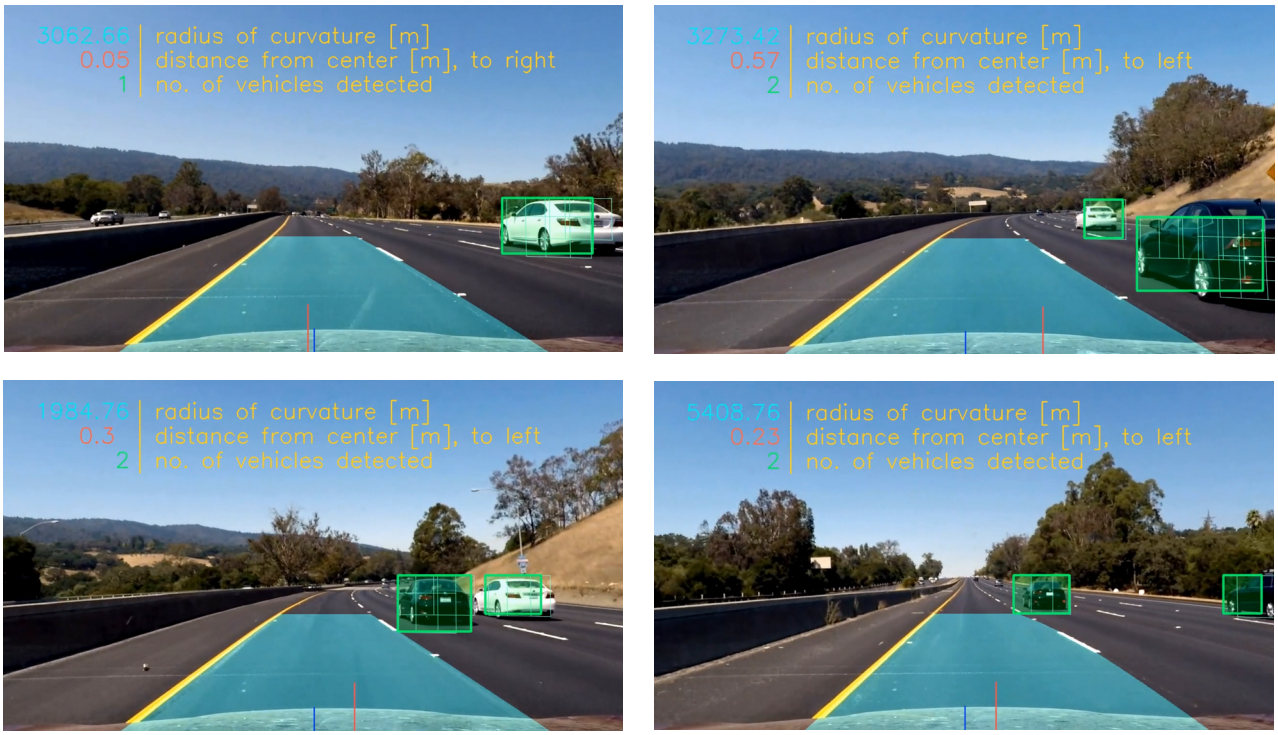


Figure 8: Screen shots of video output stream.

5 Discussion

The implementation described in this report is able to produce an output video visualizing continuous vehicle detection and tracking most of the time. In order to further improve the robustness various steps should be taken into consideration.
Working with multi-scale windows can help identifying the vehicles in different areas of the road more reliably as their size changes with distance to the camera. Especially after verifying true positives by using multi-frame accumulated heat maps (as described in the video implementation chapter) the algorithm looses track of vehicles, which are gaining distance, much faster. Another argument in favor of using multi-scale windows to avoid this effect.
This is also effected by the size and overlapping fracture of the sliding windows, which have a direct influence on the size of the output bounding boxes. Usually tight bounding boxes are what we would aim for as a precise description of the detected vehicle's location. However in certain real world situations the question might come up whether to prefer bounding boxes to end up being rather too small or too big in cases where we encounter deviations. Both ways the consequences are undesirable: a self-driving car hitting emergency brakes because of another car which was mistakenly blamed to get too close or the other way round crashing into one side of another vehicle which was detected smaller than its actual size.
While experimenting with different data, methods and parameter settings, the large data set provided by Udacity turned out to result in much more false positives in comparison to a classifier trained with a smaller set, which was used in the lessons and is derived from the large set. This seems to be quite surprising as usually more data means better training and is supposed to result in a classifier with abilities to better generalize on new input than a classifier trained on a smaller set of data. In addition the large data set does not seem to have flaws such as imbalance, meaning both types of labels are represented equally (vehicles and non-vehicles) and provides 8 times more samples than the smaller data set.
To be able to use this algorithm in real world situations the training has to be extended to an even larger data set including additional types of obstacles as for example trucks, buses, motor bikes, bicycles, pedestrians or animals like cows (indeed crucial in regions of India or Bavaria for instance).
As a last point we can improve the speed of processing a video stream by extracting HOG features once for an entire frame instead of computing HOG for every single window.

References

[1] Udacity Self-Driving Car Engineer Nanodegree Program. *Term 1, Lesson No. 22: Vehicle Detection and Tracking.*
URL: https://www.udacity.com/drive

[2] N. Dalal and B. Triggs. *Histograms of Oriented Gradients for Human Detection*. 0-7695-2372-2, June 2005.
URL: http://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf

[3] LionDoc. *RGB to Y'CrCb conversation.* Public domain, April 2012.
URL: https://commons.wikimedia.org/wiki/File:CCD.png

[4] J. Arróspide, L. Salgado, M. Nieto. *Video analysis based vehicle detection and tracking using an MCMC sampling framework.* EURASIP Journal on Advances in Signal Processing, vol. 2012, Article ID 2012:2, Jan 2012.
URL: http://www.gti.ssr.upm.es/data/Vehicle_database.html

[5] A. Geiger, P. Lenz and Raquel Urtasun. *Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite.* Conference on Computer Vision and Pattern Recognition (CVPR), April 2012.
URL: http://www.cvlibs.net/datasets/kitti/

[6] F. Pedregosa *et al. Scikit-learn: Machine Learning in Python.* JMLR 12, pp. 2825-2830, 2011.
URL: http://scikit-learn.org/stable/modules/svm.html