# Programming exercise 6: Evaluation

## Training and evaluating an object detector

We train a cascade classifier to detect profiles of cars, and then evaluate the classifiers performance by computing Intersection-over-Union (IoU) values compared to a ground truth. A cascade classifier consists of several simple classifiers (stages) that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed. These techniques are somewhat dated, but there are implementations readily available and applying them gives a useful comparison to more recent methods. For more information, you may consult the OpenCV documentation `http://docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html`.

Our method will be based on standard tools available in OpenCV, and the UIUC Image database for Car Detection, see `https://cogcomp.cs.illinois.edu/Data/Car/`. The dataset contains *positive* examples, where the desired target (car) is present, and *negative* examples, where no target is present. These examples are used to train the classifier.

## Training

**Preparing for training.** Open a terminal window, move to a subdirectory you wish to work in. Clone a repository with some useful tools that we will use for training:

```
$ git clone
https://github.com/mrnugget/opencv-haar-classifier-training.git
car_classifier
$ cd car_classifier
```

Download and extract the UIUC dataset, and move both types of training samples to their own folders, and generate a list of filenames for both.

```
$ wget http://l2r.cs.uiuc.edu/~cogcomp/Data/Car/CarData.tar.gz
$ tar -xvf CarData.tar.gz
$ mv CarData/TrainImages/pos-* ./positive_images/
$ mv CarData/TrainImages/neg-* ./negative_images/
$ find ./positive_images -iname "*.pgm" > positives.txt
$ find ./negative_images -iname "*.pgm" > negatives.txt
```

*Input/Output redirections.* Note the usage of an output redirection command `>`. This instructs to write the output of the command into a file instead of the standard output (the screen). For example, try `ls > test.txt` and inspect the output file `test.txt`. There are many more redirection types, see `https://www.tutorialspoint.com/unix/unix-io-redirections.htm`.

**Generating training data.** Apply one of the shell scripts to generate a set of training data. The script calls the `opencv_createsamples` tool multiple times with varying input arguments. Here, we create 1500 samples – you can change this and the other parameters given. Refer to OpenCV documentation at `http://docs.opencv.org/2.4/doc/user_guide/ug_traincascade.html` to determine appropriate parameter values.

```
$ perl bin/createsamples.pl positives.txt negatives.txt samples 1500
"opencv_createsamples -bgcolor 0 -bgthresh 0 -maxxangle 1.1
-maxyangle 1.1 -maxzangle 0.5 -maxidev 40 -w 25 -h 10"
```

**Training the classifier.** We then combine the training data into an appropriate format, and run the classifier.

```
$ python tools/mergevec.py -v samples/ -o samples.vec
$ opencv_traincascade -data classifier -vec samples.vec
-bg negatives.txt -numStages 10
-acceptanceRatioBreakValue 10e-5 -numPos 1000
-numNeg 600 -w 25 -h 10 -mode ALL
-precalcValBufSize 2048  -precalcIdxBufSize 2048
```

1. Refer to the previous link for choosing appropriate arguments for `opencv_traincascade`.
2. Save a note on the parameters you used in training to a file somewhere.

The classifier training will take some minutes, and the output will be written to the subdirectory `classifier`.

## Classification

We now run our trained classifier on a set of test data. Download and extract the provided archive `cascadedet.tar.gz` from MIN-CommSy. Set up a `build` folder and compile the software.

1. Study how the executable is called from the command line.
2. Try to run it with the classifier you trained on one of the test images in `CarData/TestImages`, showing the output image.
3. Check the output and determine how it corresponds to the image and bounding boxes shown.
4. Disable the showing of the output image, and redirect the output of the command to a text file.

It is tedious to call the classifier separately for every test image. Write a bash script to batch process every file in a given folder. Create a file called `classify.sh` and insert the following contents to it.

```
#!/bin/bash
# Example usage: . classify.sh path/to/some_directory
# $1 refers to the first input argument given by the user
FILES=`ls $1/*.pgm`
for f in $FILES
do
  echo "Processing file $f "
  # Variable $f stores current file name
  # Example (replace by call to classifier)
  cat $f
done
```

1. Modify `classify.sh` to call the classifier software for every test image in a given folder.
2. The output of the classifier is multiple lines in the format `[(1,2), (3,4)]`. Filter the output to the format `1 2 3 4`. This will make reading the data easier later.
3. Finally, redirect the filtered output to a text file, one for each test image.
4. Run the script to apply the classifier to every test image. The script can be executed as follows (note the dot and space):
   `. classify.sh <testdata-directory>`

Hints:

- For filtering, use redirects to e.g. the classical UNIX command line tools such as `grep`, `sed`, or `tr`.
- Multiple redirects can be combined, e.g.
  `echo Hello world! | sed 's/[el]/a/g' | tr -d "o" > mytext.txt`

## Evaluation

We evaluate the classifier by comparing its output to a known ground truth using the IoU metric (see Figures 1 and 2).

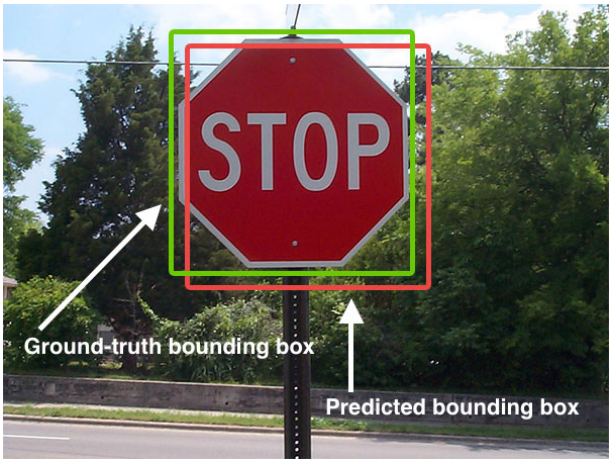


Figure 1: Prediction and ground truth.

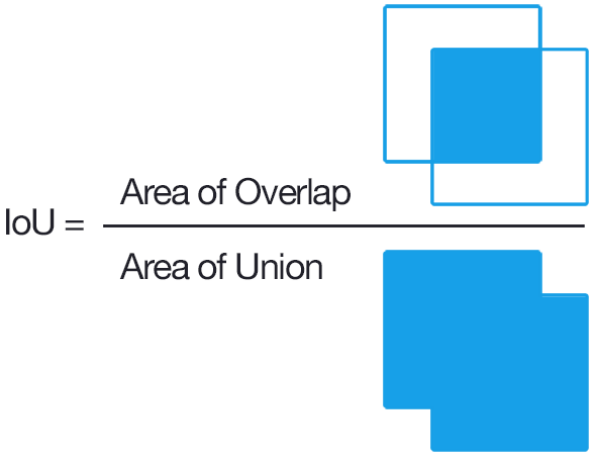


Figure 2: IoU.

**Understanding and preprocessing the ground truth data.** Ground truth for the test images in the Car Dataset are in the file `CarData/trueLocations.txt`. The ground truth bounding boxes are all of size 100 by 40 pixels. Each line in this file has the following format:

```
number: (car1_top_left_corner_x, car1_top_left_corner_y)
... (carN_top_left_corner_x, carN_top_left_corner_y)
```

which indicates that in test image `number` there are $N$ cars, with a 100-by-40 bounding boxes whose top left corners are specified. In other words, the bottom right corner coordinates of bounding boxes are given by `(car1_top_left_corner_x + 100, car1_top_left_corner_y + 40)`

Apply some text processing magic to 1) remove the leading `number:` and parentheses and 2) split the monolithic ground truth text file into a set of files, one containing ground truth for one test image:

```
$ mkdir ground_truth && cd ground_truth
$ cut -d : -f 2 ../CarData/trueLocations.txt |
sed -e 's/[()]//g' -e 's/,/ /g' -e 's/ //' |
split -l 1 -a 3 --additional-suffix=.txt -d - test-
```

**Evaluation.** You should now have a set of .txt files for the prediction outputs of your classifier, and for the ground truth. Download the Python script `iou_calc.py` from MIN-CommSy.

1. Study how the script works. Fill in the missing parts inside the main loop to complete it.
2. Write a shell script to call the IOU script for every pair of classifier outputs - ground truth data text files. **Do not** redirect every output to a separate text file. Rather, use the redirect `>>` to append each output to the end of a single output file.
3. Inspect the IOU results (e.g., using another Python script). Compute the average and standard deviation of IOU.

Hints and notes:

- Note that in the Python script we only output the best IOU value among all pairs of ground truth and prediction bounding boxes. This makes the classifier seem a bit better than it actually is.
- Can you improve the performance of the classifier by retraining with another set of parameters? How would you judge if improvement happened or not?