Kamalnath Devarajan

UH ID:1906161

COSC 6373 – HW3 Report

This report describes how I managed to build a parking lot vacancy detector.

The Problem:

A public infrastructure has various parking lots. The parking lots get completely occupied very often and the public visiting the infrastructure spend too much time looking for a parking space, unaware that the parking lot is completely occupied. They would like to implement an automated solution to convey this information by displaying the number of available parking spaces at the entrance to the parking lot.

These parking lots are overlooked be surveillance cameras. The task is to leverage them to detect and count the number of empty parking spots.

The Solution:

We'll need to build and train a classifier which detects if a car is present at a parking spot or not. It consists of four steps:

1) Create a training dataset
2) Train a cascade classifier based on HAAR and Local Binary Patterns (LBP) features
3) Car Detection – using the trained classifier to detect cars in a test image.
4) Parking Spot Analysis – Use the detected cars to decide if a parking spot is vacant or not.

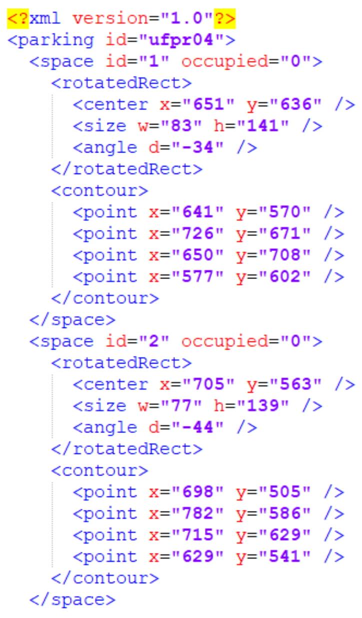
Dataset

The dataset contains images from three different parking lots, namely, parking1a, parking1b and parking2. Each parking lot has images under three different weather conditions, sunny, rainy and cloudy. Each parking lot image in the dataset is supplemented with an xml file, which provides information about each parking spot in the parking lot, its occupancy status, the contours of the parking spot, etc. We're going to train our classifier using all the pictures of the parking lots under the sunny scenario and test it on images of parking lots under rainy and cloudy conditions.

Procedure

1) Creation of the training dataset

This dataset must contain images of a single object, (in this case, a car) because we want to detect if a car is present at a parking spot or not. What we have are images of the parking lots. Hence, we must crop out images of cars from each of the parking lot images. The xml file of each image provides information about the contours of each parking spot and whether its occupied or not.



```xml
<?xml version="1.0"?>
<parking id="ufpr04">
  <space id="1" occupied="0">
    <rotatedRect>
      <center x="651" y="636" />
      <size w="83" h="141" />
      <angle d="-34" />
    </rotatedRect>
    <contour>
      <point x="641" y="570" />
      <point x="726" y="671" />
      <point x="650" y="708" />
      <point x="577" y="602" />
    </contour>
  </space>
  <space id="2" occupied="0">
    <rotatedRect>
      <center x="705" y="563" />
      <size w="77" h="139" />
      <angle d="-44" />
    </rotatedRect>
    <contour>
      <point x="698" y="505" />
      <point x="782" y="586" />
      <point x="715" y="629" />
      <point x="629" y="541" />
    </contour>
  </space>
```

Fig 1: Description of the ground-truth xml file

We can use this piece of information to extract the images of cars from the parking lot images.

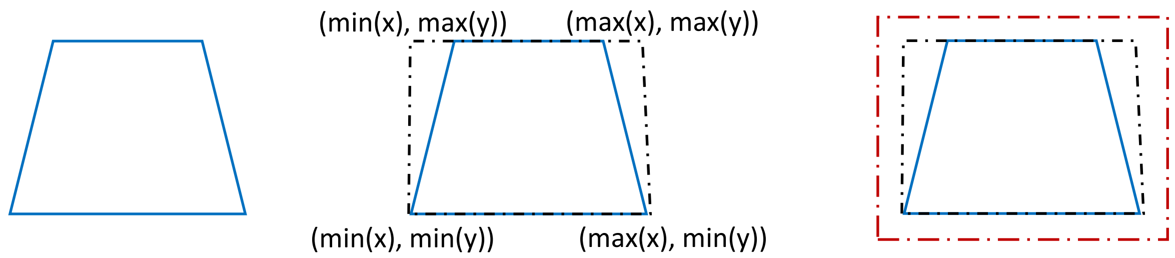
The program pos_img_train.py reads all the xml files and its corresponding jpg files in the sunny subdirectories of all three parking lot directories, to extract cropped car images. I used *xml.etree.ElementTree* to parse the xml file as well as the *glob* library.

Referring to the figure above, we need to extract the four points under *contour*. This will tell us where the car is present in the image. If looked upon closely, the points clearly don't form a rectangle, but they do form some sort of quadrilateral. We'll need to transform that quadrilateral into a rectangle. In order to do that, we'll need to make a few assumptions.

Let x be the set of x-coordinates of the contours of a parking spot and y be the set of y-coordinates of the contours of the same parking spot.

If we assume the coordinates of one of the vertices of the quadrilateral to be `(min(x), min(y))`, then the coordinates of the vertex opposite to it will be `(max(x), max(y))`.

If we take the example of space id = 1 from the above figure (Fig 1), the opposite vertices have co-ordinates (577, 570) and (726, 708). The other two vertices will be (726, 708) and (577, 708). Hence, we get a rectangle. I extended the boundaries by 10 pixels over each side, to ensure that the entire car is enclosed within the rectangle (Refer Fig 2 below). With the dimensions of the rectangle enclosing the car known, each and every car is cropped from all the parking lot images under the sunny sub-directories.



Bounding box of the parking spot *Transforming the quadrilateral into a rectangle* *Extending the boundaries by 10 pixels on all sides*

Fig 2: Logic to generate cropped positive images

Every cropped car image is stored in a folder named *positive_images*. I obtained a total of 184,501 cropped car images. Few examples of positive images are shown below:

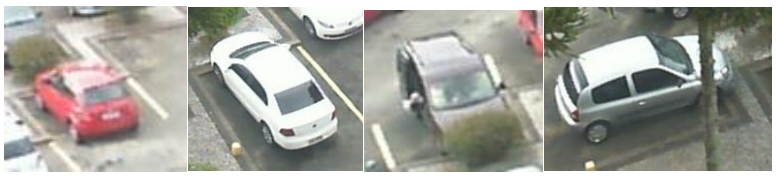


Fig 3: Examples of positive images, in this case, it's a car.

Next, we need a dataset of negative images (images which don't have a car in it). I collected images from image data sets that are publicly available on the internet.

- 256 Object Categories: http://www.vision.caltech.edu/Image_Datasets/Caltech256/
- Labelled Faces in the Wild: http://vis-www.cs.umass.edu/lfw/
- Food-101 Dataset: https://www.kaggle.com/kmader/food41

Few examples of negative images are shown below:

Fig 4: Examples of negative images

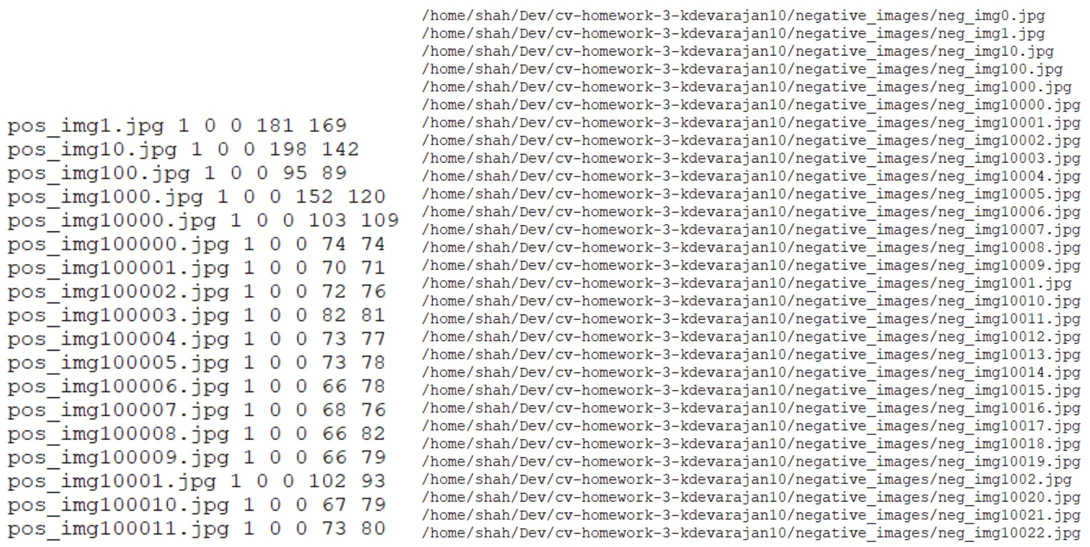
I stored a total of 93,429 negative images in a directory named, *negative_images.*

Next, we need to create files, called *bg.txt* and *info.dat* which contain the path of every negative image and positive image, respectively. I followed the instructions issued by the OpenCV documentation on cascade classifier training to create the files. https://docs.opencv.org/4.2.0/dc/d88/tutorial_traincascade.html

I had to make a few diversions from the instructions given in the above documentation. The *info.dat* file shouldn't contain the absolute path of the cropped positive images. It should contain the image filename alone, along with the number of object instances and the co-ordinates of the object bounding box, separated by spaces.

On the other hand, for *bg.txt*, make sure it contains the **absolute path** and not the relative path of each negative image.

The bg.txt and info.dat need to be placed in the same directory as the positive and negative images respectively, i.e., in *positive_images* and *negative_images* respectively.



```
pos_img1.jpg 1 0 0 181 169
pos_img10.jpg 1 0 0 198 142
pos_img100.jpg 1 0 0 95 89
pos_img1000.jpg 1 0 0 152 120
pos_img10000.jpg 1 0 0 103 109
pos_img100000.jpg 1 0 0 74 74
pos_img100001.jpg 1 0 0 70 71
pos_img100002.jpg 1 0 0 72 76
pos_img100003.jpg 1 0 0 82 81
pos_img100004.jpg 1 0 0 73 77
pos_img100005.jpg 1 0 0 73 78
pos_img100006.jpg 1 0 0 66 78
pos_img100007.jpg 1 0 0 68 76
pos_img100008.jpg 1 0 0 66 82
pos_img100009.jpg 1 0 0 66 79
pos_img10001.jpg 1 0 0 102 93
pos_img100010.jpg 1 0 0 67 79
pos_img100011.jpg 1 0 0 73 80
```

```
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img0.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img1.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img100.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img1000.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10000.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10001.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10002.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10003.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10004.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10005.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10006.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10007.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10008.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10009.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img1001.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10010.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10011.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10012.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10013.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10014.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10015.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10016.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10017.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10018.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10019.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img1002.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10020.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10021.jpg
/home/shah/Dev/cv-homework-3-kdevarajan10/negative_images/neg_img10022.jpg
```

Fig 5: Contents of info.dat and bg.txt files

Next, we need to create positive samples using the *opencv_createsamples* application, with which we would train the cascade classifier.

The application creates a .vec file of *-num* positive samples, which we feed in as an argument. I didn't feed my entire dataset of positive images to *opencv_createsamples* application, with overfitting in mind. The classifier would have done very well on training set, but would have performed poorly on the test set. Hence, I chose 75,000 as the number of positive samples to be created.

```
opencv_createsamples -info positive_images/info.dat -vec positive.vec
-num 75000 -w 24 -h 24
```

This will create 75,000 positive samples, each of width and height of 24 pixels and stored in *positive.vec*.

The width and height of the training window size is 24x24. Larger values of width and height are not used since it would cause memory allocation problems, causing the system to crash and RAM to run out. Besides, a larger window size would take even longer to train. Hence, I picked small dimensions of 24x24 as width and height of each positive sample.

2) Training the cascade classifier

Once the samples are created, we can train our classifier using the *opencv_traincascade* application.

First, I trained an LBP classifier. I trained one with 45,000 positive samples and 50,000 negative samples.

There is no obvious rule for positive/negative samples ratio. Usually, it's advised to take more negative samples than positive ones, but never less than that. This is because any classifier will give way more negative responses than positive ones on a common scenario.

I left the minimum hit rate and maximum false alarm rate at their default values. I chose 8 stages to train the LBP classifier.

It took **3 hours and 26 minutes and 28 seconds** to train the LBP classifier for 8 stages with 45,000 positive samples and 50,000 negative samples. The following command initiates training of an LBP classifier:

```
opencv_traincascade -data LBP_training/ -vec positive.vec -bg
bg_orig1.txt -numPos 45000 -numNeg 50000 -numStages 8 -featureType
LBP -w 24 -h 24
```

Similarly, for training the HAAR classifier, it took **16 hours 28 minutes and 47 seconds** for 6 stages with 10,000 positive samples and 12,000 negative samples. The following command initiates the training of a HAAR classifier:

```
opencv_traincascade -data haar_training/ -vec positive.vec -bg
bg_orig1.txt -numPos 10000 -numNeg 12000 -numStages 6 -featureType
HAAR -w 24 -h 24
```

Notice that the HAAR classifier takes much longer to train than the LBP classifier.

Once training is done, an xml file called *cascade.xml* is generated in both the folders, *LBP_training* and *haar_training*.

Now that the model has been trained, we can test our model using the images from the cloudy and rainy sub-directories.

3) Car Detection

The python file, *testing_cars.py* performs car detection, taking the type of classifier and the image as input parameters, along with two other important parameters which heavily affect the accuracy of the classifier: *scaleFactor* and *minNeighbors*

If the value of the scale factor is too large, then only a few layers of the image pyramid is evaluated, which implies that we'll be missing out on detecting cars at scales in between the layers of the image pyramid. If it's the value of scale factor is too less, then we'll be scanning more carefully, hence detecting more cars in the process. However, there is a trade-off between the duration of detection and the value of the scale factor. Smaller the scale factor, more is the time taken to detect cars and vice versa. If the scale factor is smaller, there's a higher chance in the increase of false-positives (Detecting a car, when there wasn't one in the first place).

The other parameter is *minNeighbors*, which tells us the minimum number of detected bounding boxes in a given area in the image, to be considered a car. This parameter is useful in removing redundant detected bounding boxes, or in other words, reducing the number of false-positives.

Finding the ideal value for *scaleFactor* and *minNeighbors* is a difficult task, since its values vary from image to image. As future scope, one can train these parameters too and fine-tune them to get the optimal result.

After trial and error, I figured out that ***scaleFactor* = 1.33 and *minNeighbors* = 20**, gave decent results overall for most of the images.

```
python   testing_cars.py   -img   cloudy/2013-01-16_07_50_03.jpg   -ft
HAAR/LBP -sf 1.33 -mn 20
```

Few results of car detection are shown below:

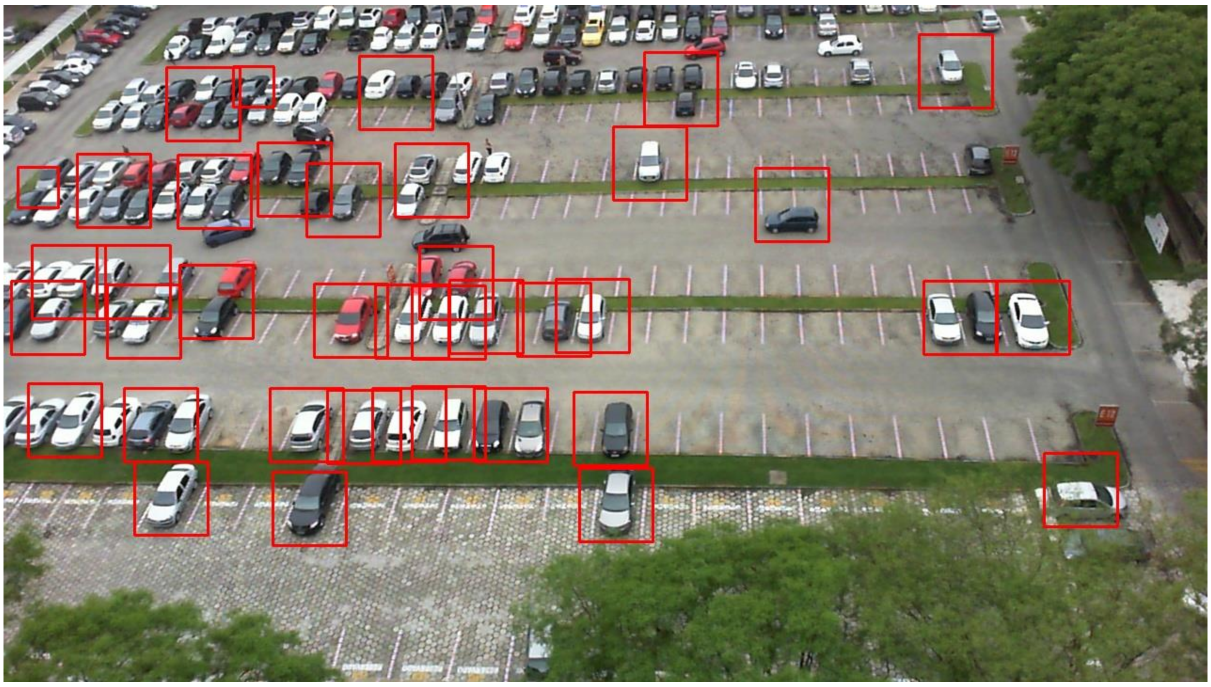


Fig 6: HAAR Classifier, *scaleFactor* = 1.8, *minNeighbors* = 15

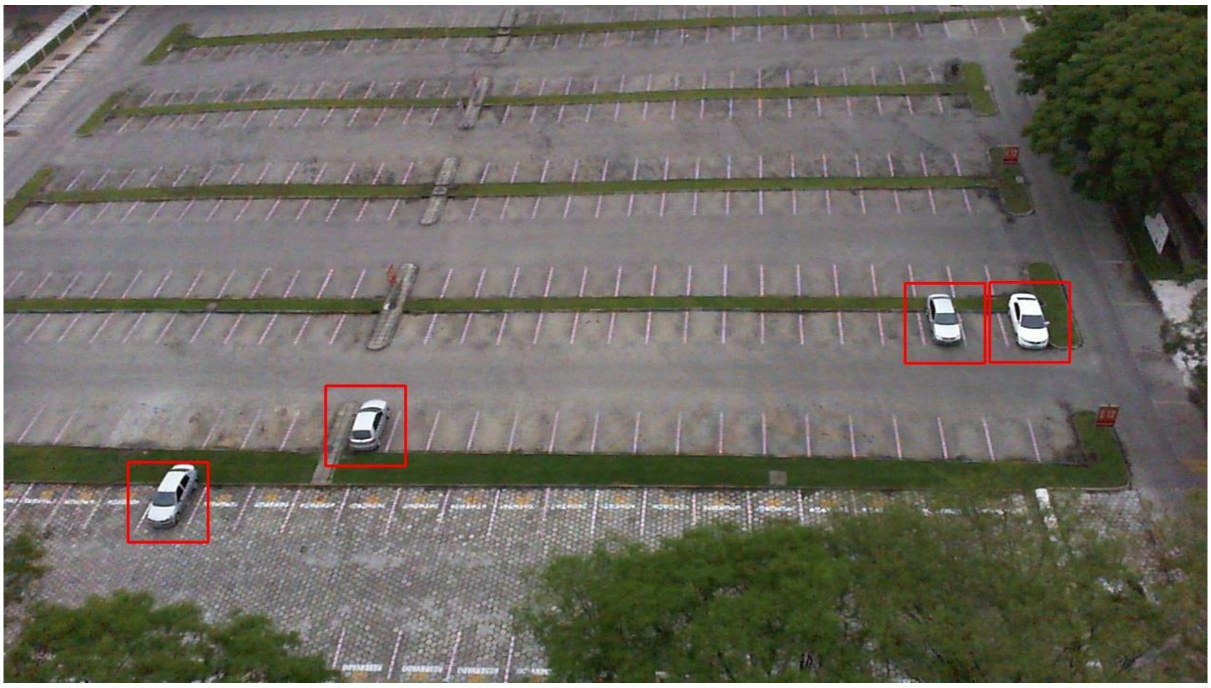


Fig 7: LBP Classifier, *scaleFactor* = 1.885, *minNeighbors* = 19

4) Parking Spot Analysis

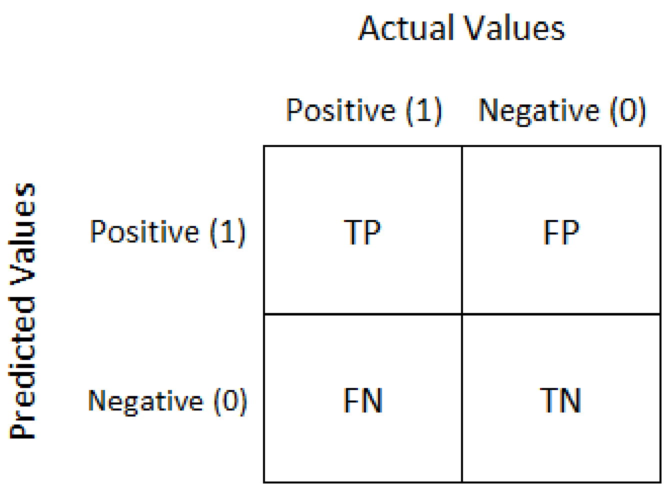
Now that our classifier is able to detect cars from an image, we can finally build our parking lot detector and count the number of parking spots that are occupied and empty.

In order to find out if a parking spot is occupied or not, the logic I used is fairly simple. A rectangle is detected if a car is present. We also have the contours of each parking spot and its occupancy status from the ground-truth xml file.

If the detected rectangle encloses the center of the bounding box of the parking spot and its occupancy status is '1', that means a car is present at that particular parking spot and it has been detected by the classifier (true-positive).

On the other hand, if the detected rectangle encloses the center of the bounding box of the parking spot but its occupancy status is '0', that means a car is absent at that parking spot, but a car has been mistakenly detected by the classifier (false-positive).

The figure below describes a confusion matrix which ought to make it easier to understand the classification process.



		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

Fig 8: Confusion Matrix

From earlier in the report, I described how we derived the co-ordinates of each parking spot, each vertex of the parking spot involving a combination of the values `(max(x), max(y), min(x), min(y))`. From the vertices, we can calculate the center of the rectangle as `((min(x) + max(x))/2, (min(y) + max(y))/2)`. The figure below explains the above logic more clearly.

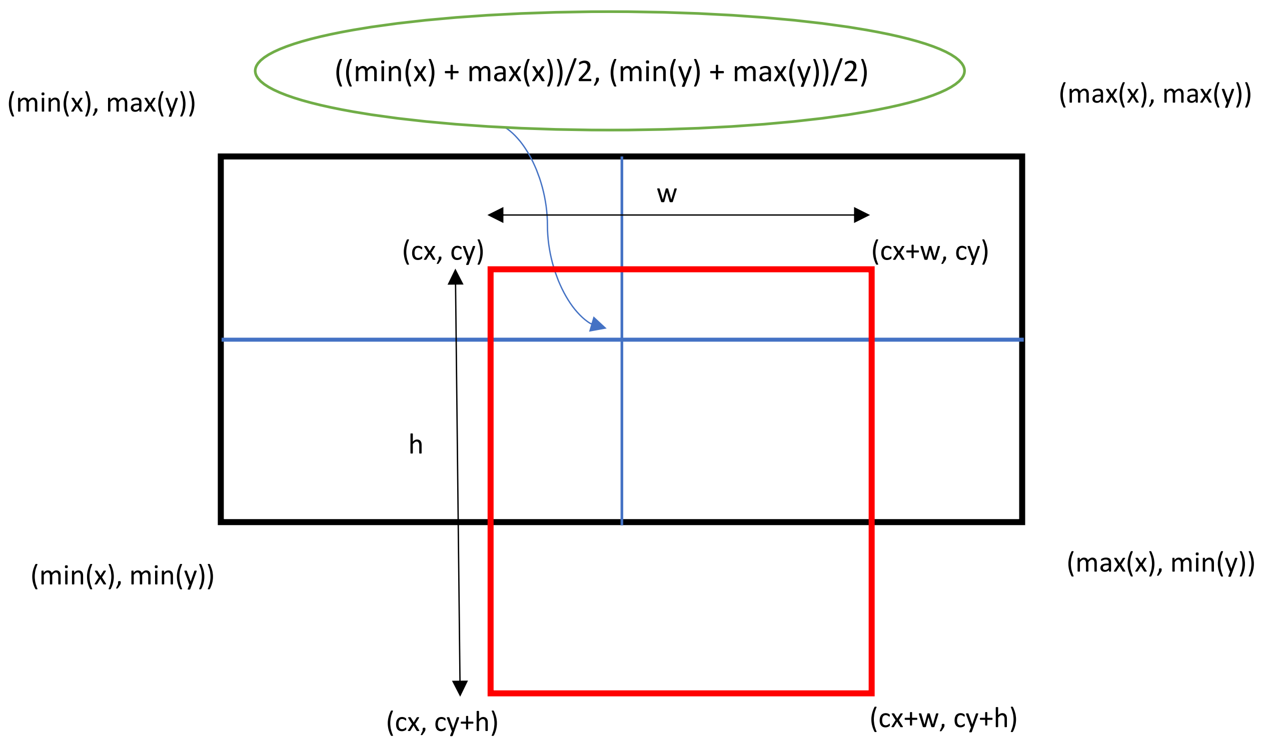


Fig 9: Logic to determine if the parking spot is occupied or not.

Now, we have the number of true positives and false positives in an image. The accuracy of a classifier is defined as follows:

```
Accuracy = ((#True Positives + #True Negatives) / (#True Positives + #True
         Negatives + #False Positives + #False Negatives)) x 100
```

We require the number of true negatives and false negatives to calculate the accuracy. They can be evaluated as follows:

```
#True Negatives = #Parking spots correctly classified as empty by the
                  Classifier

                = #Empty parking spots – #Parking spots incorrectly
                  classified as empty by the classifier

                = #Empty parking spots – #False Positives

#False Negatives = #Parking spots incorrectly classified as occupied by
                   the classifier

                 = #Occupied parking spots – #Parking spots correctly
                   classified as occupied by the classifier
                 = #Occupied parking spots – #True Positives
```

The number of empty and occupied parking spots can be calculated from the ground-truth XML file.

Results:

The calculations above are only with respect to the annotated parking spots in the ground-truth xml file. If there are multiple cars at parking spots which are not annotated, they are ignored in our calculations.

I tested the classifier on 25 images from rainy scenario and 25 images from the cloudy scenario. The table below summarizes the performance of the classifier tested on the 50 images.

Sl. No.	Image Name	Classifier Feature	Training Stages	#Positives	#Negatives	TP	FP	Accuracy
1	2012-10-26-_07_39_28.jpg	HAAR	6	10000	12000	21	6	83.0
2	2013-03-19_06_35_00.jpg	HAAR	6	10000	12000	0	5	87.5
3	2013-01-16_08_35_04.jpg	LBP	8	45000	50000	0	0	100.0
4	2012-10-26-_07_29_28.jpg	HAAR	6	10000	12000	15	5	88.0
5	2013-03-19_06_45_00.jpg	HAAR	6	10000	12000	0	1	97.5
6	2013-01-16_11_25_07.jpg	LBP	8	45000	50000	0	0	64.286
7	2013-01-16_08_30_04.jpg	HAAR	6	10000	12000	0	4	85.714
8	2012-10-26-_07_44_28.jpg	LBP	8	45000	50000	36	3	91.0
9	2013-01-16_07_55_03.jpg	LBP	8	45000	50000	0	0	100.0
10	2013-03-19_07_00_01.jpg	LBP	8	45000	50000	1	2	90.0
11	2012-10-26-_07_24_27.jpg	LBP	8	45000	50000	19	13	87.0
12	2013-03-19_06_40_00.jpg	LBP	8	45000	50000	0	3	92.5
13	2012-10-26-_06_44_25.jpg	LBP	8	45000	50000	6	17	83.0
14	2013-03-19_06_30_00.jpg	LBP	8	45000	50000	0	0	100.0
15	2013-03-19_07_30_01.jpg	LBP	8	45000	50000	16	3	65.0

16	2012-10-26-_07_19_27.jpg	HAAR	6	10000	12000	13	9	87.0
17	2012-10-26-_06_39_25.jpg	HAAR	6	10000	12000	4	8	92.0
18	2013-01-16_11_50_07.jpg	HAAR	6	10000	12000	7	2	82.143
19	2013-01-16_11_10_06.jpg	HAAR	6	10000	12000	7	5	67.857
20	2012-10-26-_06_34_24.jpg	LBP	8	45000	50000	4	14	86.0
21	2013-01-16_11_00_06.jpg	HAAR	6	10000	12000	5	2	75.0
22	2013-03-19_06_55_01.jpg	HAAR	6	10000	12000	2	3	92.5
23	2013-01-16_11_40_07.jpg	HAAR	6	10000	12000	5	3	71.429
24	2012-10-26-_06_29_24.jpg	HAAR	6	10000	12000	4	7	93.0
25	2013-01-16_10_45_06.jpg	LBP	8	45000	50000	1	1	67.857
26	2012-10-26-_07_34_28.jpg	LBP	8	45000	50000	26	14	84.0
27	2013-03-19_09_15_03.jpg	HAAR	6	10000	12000	27	0	70.0
28	2013-01-16_08_15_03.jpg	LBP	8	45000	50000	0	0	100.0
29	2013-01-16_11_45_07.jpg	LBP	8	45000	50000	2	3	64.286
30	2013-01-16_07_50_03.jpg	HAAR	6	10000	12000	0	3	89.253
31	2013-03-19_07_05_01.jpg	HAAR	6	10000	12000	6	4	90.0
32	2012-10-26-_07_49_29.jpg	HAAR	6	10000	12000	44	4	88.0
33	2013-01-16_11_35_07.jpg	LBP	8	45000	50000	2	3	60.714
34	2013-01-16_08_05_03.jpg	LBP	8	45000	50000	0	0	100.0
35	2013-01-16_11_05_06.jpg	LBP	8	45000	50000	2	1	67.857
36	2013-03-19_12_45_07.jpg	LBP	8	45000	50000	23	0	60.0
37	2013-01-16_11_30_07.jpg	HAAR	6	10000	12000	4	3	67.857
38	2012-10-26-_06_24_24.jpg	LBP	8	45000	50000	4	10	90.0

39	2013-01-16_11_20_07.jpg	HAAR	6	10000	12000	7	3	75.0
40	2013-01-16_10_50_06.jpg	HAAR	6	10000	12000	6	5	71.429
41	2013-01-16_08_20_03.jpg	HAAR	6	10000	12000	0	8	71.429
42	2013-01-16_11_15_07.jpg	LBP	8	45000	50000	1	0	67.857
43	2013-01-16_08_00_03.jpg	HAAR	6	10000	12000	0	5	82.143
44	2013-03-19_06_50_01.jpg	LBP	8	45000	50000	0	0	100.0
45	2013-01-16_10_55_06.jpg	LBP	8	45000	50000	7	5	71.429
46	2013-01-16_08_25_03.jpg	LBP	8	45000	50000	0	0	100.0
47	2013-03-19_07_25_01.jpg	HAAR	6	10000	12000	18	2	80.0
48	2013-01-16_08_10_03.jpg	HAAR	6	10000	12000	0	5	82.143
49	2013-01-16_10_40_06.jpg	HAAR	6	10000	12000	6	3	75.0
50	2013-03-19_07_10_01.jpg	LBP	8	45000	50000	7	3	85.0

Discussion of Results and Inference:

I inferred a couple of things from the assignment. HAAR training takes extremely long to train. Instead of training from scratch, one could utilize transfer learning instead which would save lots of time.

As I remarked earlier, there is no fixed value for *minNeighbors* and *scaleFactor* which would work well for all images. Only by trial and error, I was able to obtain reasonable values for the two parameters to get a good accuracy. For example, for a certain value of *minNeighbors* and *scaleFactor*, if the accuracy was around 50% and if the number of false positives was on the higher side, one could increase the *minNeighbors* to prune out the number of false positives.

If we take a look at the two images below, the first one was tested by a HAAR classifier with *minNeighbors* = 20 and *scaleFactor* = 1.33. I obtained an accuracy of 81%.

The second one was also tested by a HAAR classifier with the same *scaleFactor* value but *minNeighbors* = 35. Here, I obtained an accuracy of 87%.

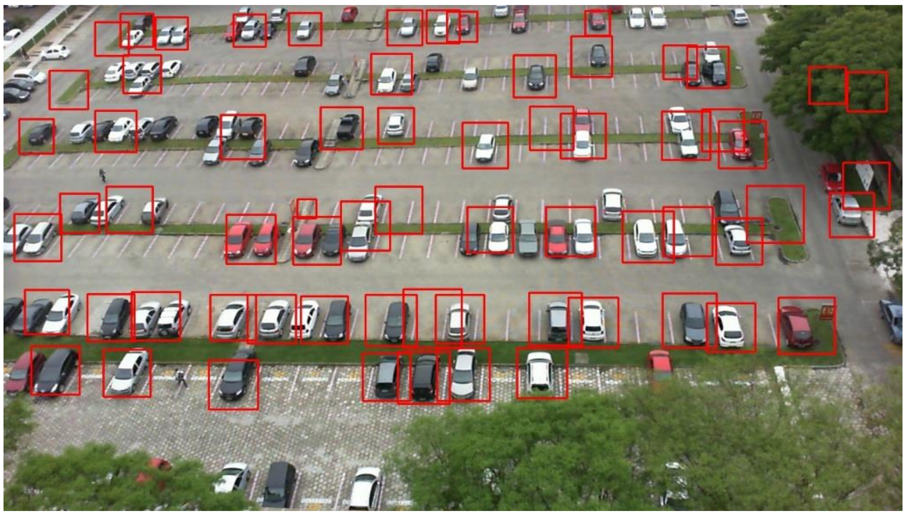


Fig 10: Image tested on HAAR Classifier, *scaleFactor* = 1.33, *minNeighbors* = 20

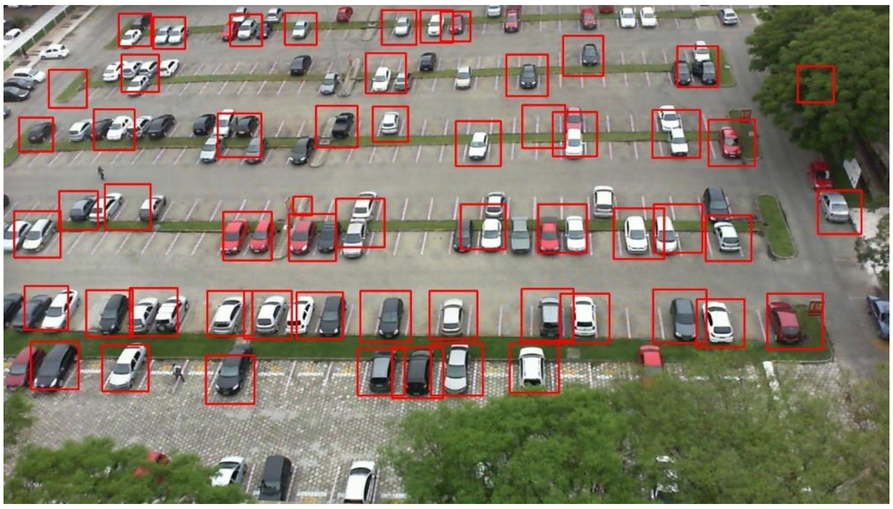


Fig 10: The same image tested on HAAR Classifier, *scaleFactor* = 1.33, *minNeighbors* = 35

```
(cv) (shahdev) shah@d4ec59b7f625:~/Dev/cv-homework-3-kdevarajan10$ python
testing_cars.py -img 2012-10-25_12_38_32.jpg -ft HAAR -sf 1.33 -mn 20

Image Filename:  2012-10-25_12_38_32.jpg

The number of parking spots that are filled =  44
The number of parking spots that are empty =  56


The number of True Positives =  38
The number of False Positives =  13
The number of True Negatives =  43The number of False Negatives =  6


Accuracy =  81.0
```

```
(cv) (shahdev) shah@d4ec59b7f625:~/Dev/cv-homework-3-kdevarajan10$ python
testing_cars.py -img 2012-10-25_12_38_32.jpg -ft HAAR -sf 1.33 -mn 35

Image Filename:  2012-10-25_12_38_32.jpg

The number of parking spots that are filled =  44
The number of parking spots that are empty =  56


The number of True Positives =  38
The number of False Positives =  7
The number of True Negatives =  49
The number of False Negatives =  6


Accuracy = 87.0
```

From the terminal outputs, we can see that the number of false positives reduced from 13 to 7. Also, the number of true negatives went up from 43 to 49. This is due to increasing the value of *minNeighbors*. If you also take a look at the two output images of the parking lots above, redundant detected rectangles are removed.

Similarly, further experimentations with the scaleFactor also yield similar increases in accuracy.