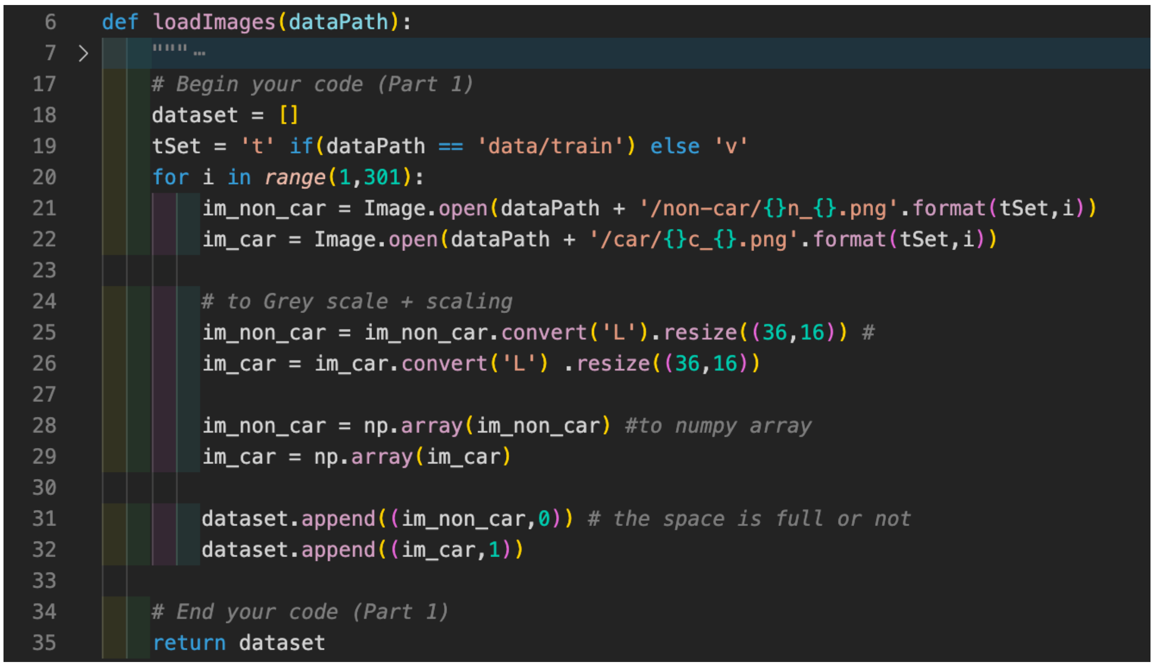
HW1 Report_0819818 許慈若

Section I

In this section, the codes that I've modified or written will be explained. An emphasis will be placed on the implementation of the algorithm AdaBoost.

1. Load and prepare the dataset



```python
def loadImages(dataPath):
    """…
    # Begin your code (Part 1)
    dataset = []
    tSet = 't' if(dataPath == 'data/train') else 'v'
    for i in range(1,301):
        im_non_car = Image.open(dataPath + '/non-car/{}n_{}.png'.format(tSet,i))
        im_car = Image.open(dataPath + '/car/{}c_{}.png'.format(tSet,i))

        # to Grey scale + scaling
        im_non_car = im_non_car.convert('L').resize((36,16)) #
        im_car = im_car.convert('L') .resize((36,16))

        im_non_car = np.array(im_non_car) #to numpy array
        im_car = np.array(im_car)

        dataset.append((im_non_car,0)) # the space is full or not
        dataset.append((im_car,1))

    # End your code (Part 1)
    return dataset
```

Depending on whether it's training or testing data, the function will (1) load the data, (2) convert into grey scale, (3) resize the images into (36, 16) as required via the package "Pillow". Images and their labels will be appended to the **dataset** list, and finally, be returned.

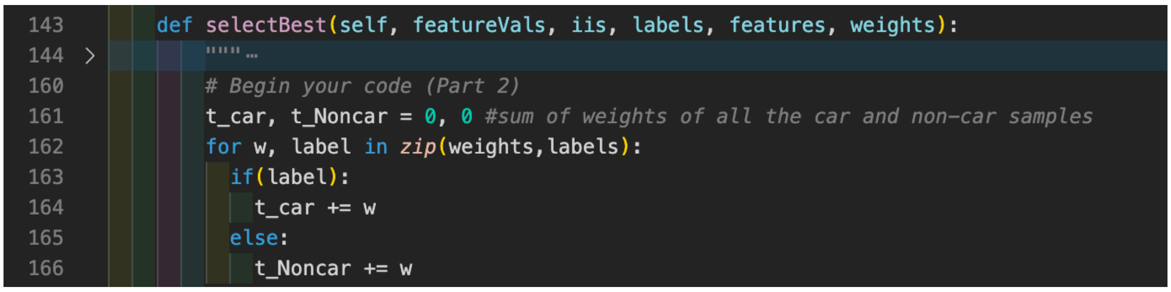
2. Implement AdaBoost algorithm

I will briefly explain the concept and implementation of AdaBoost, following the structure of Adaboost.**train** function:

Since the specific features can appear at different positions and scales, we utilize tricks from Viola Jones algorithm by first calculating integral images (a summed-area table) to build Harr like features more efficiently. After obtaining the features, we add the features to our training data, which are in the form of integral images. Then, we just initialize our weights equally to be 1/600.

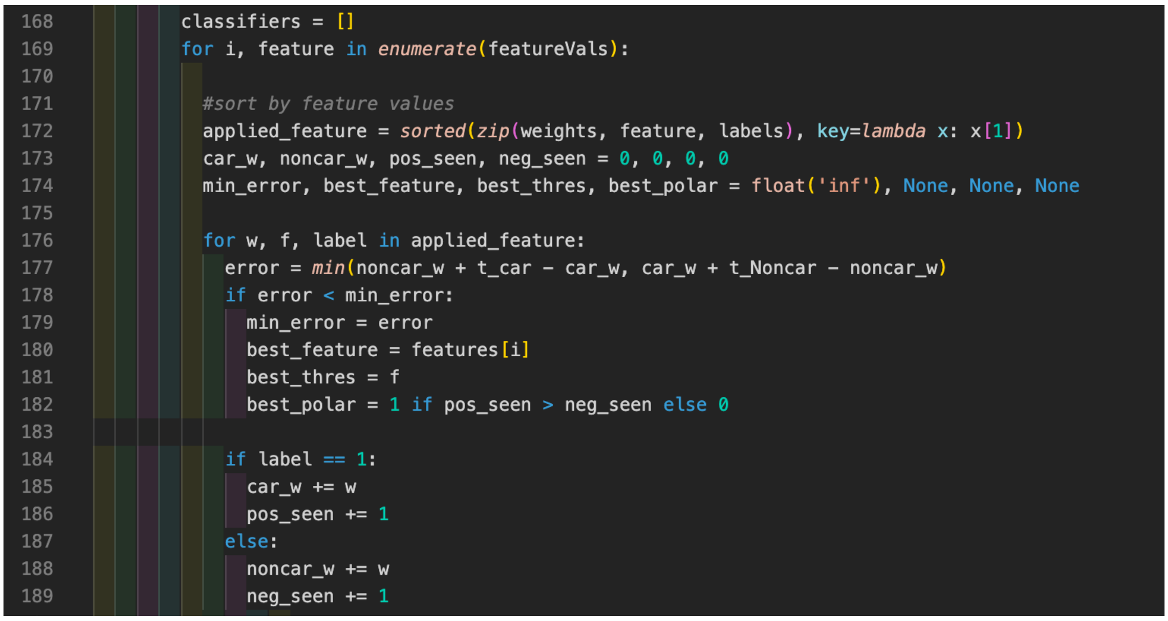
To decide which features can be part of the final classifier, here comes in the core of AdaBoost algorithm. AdaBoost stands for adaptive boosting, it iteratively selects an ensemble of component (weak) classifiers. These weak classifiers only need to split training examples for at least 50% accuracy, but eventually a sequence of weak classifiers is combined into a strong one. During the training, we reweight the misclassified examples iteratively, and that gives an intuition to "adaptive".

For each iteration, we normalize our weights first. Then, the **selectBest** function aims to return the best weak classifier for the current given weights.



```python
143     def selectBest(self, featureVals, iis, labels, features, weights):
144 >       """…
160         # Begin your code (Part 2)
161         t_car, t_Noncar = 0, 0 #sum of weights of all the car and non-car samples
162         for w, label in zip(weights,labels):
163           if(label):
164             t_car += w
165           else:
166             t_Noncar += w
```

Add up the positive sample weights and negative sample weights according to their labels.



```python
168         classifiers = []
169         for i, feature in enumerate(featureVals):
170 
171           #sort by feature values
172           applied_feature = sorted(zip(weights, feature, labels), key=lambda x: x[1])
173           car_w, noncar_w, pos_seen, neg_seen = 0, 0, 0, 0
174           min_error, best_feature, best_thres, best_polar = float('inf'), None, None, None
175 
176           for w, f, label in applied_feature:
177             error = min(noncar_w + t_car - car_w, car_w + t_Noncar - noncar_w)
178             if error < min_error:
179               min_error = error
180               best_feature = features[i]
181               best_thres = f
182               best_polar = 1 if pos_seen > neg_seen else 0
183 
184             if label == 1:
185               car_w += w
186               pos_seen += 1
187             else:
188               noncar_w += w
189               neg_seen += 1
```

In this part, the goal is to train a classifier for each feature.

[172] Having evaluated each filter for every example(featureVals), we sort these features by their values to pick the threshold efficiently.

[176~189] At each applied feature we compute their errors:

$$error = \min\left(NonCar\_w + (T\_car - Car\_w), \quad Car\_w + (T\_NonCar - NonCar\_w)\right)$$

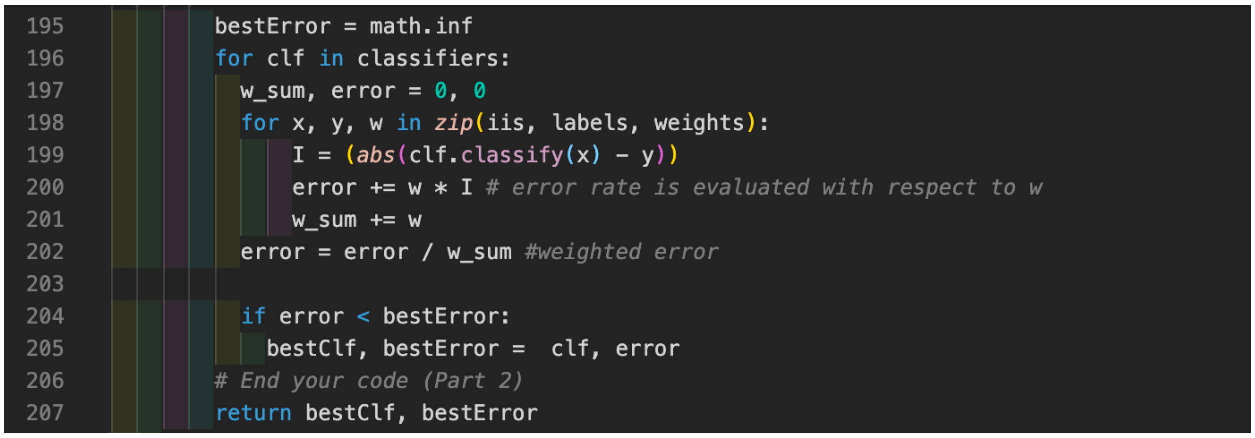
, where:

(1) **NonCar_w** and **Car_w** stand for the sum of weights for car and non-car samples "so far" respectively.

(2) **T_Car** and **T_NonCar** stand for the sum of weights of all the car and non-car samples respectively.

With this minimum error, we take the corresponding feature value as the best threshold for that classifier, and also set the polarity accordingly. Then we keep track of **Car_w**/ **NonCar_w** and **pos_seen**/**neg_seen** depending on their labels.

[192~193] Add the chosen classifier to the classifiers list.



```
195 bestError = math.inf
196 for clf in classifiers:
197     w_sum, error = 0, 0
198     for x, y, w in zip(iis, labels, weights):
199         I = (abs(clf.classify(x) - y))
200         error += w * I # error rate is evaluated with respect to w
201         w_sum += w
202     error = error / w_sum #weighted error
203 
204     if error < bestError:
205         bestClf, bestError =  clf, error
206 # End your code (Part 2)
207 return bestClf, bestError
```

For every candidate weak classifier from the classifier lists we just obtained, we estimate the (weighted) error of this classifier on the processed training data (the integral images). And we return the classifier with the minimum error:

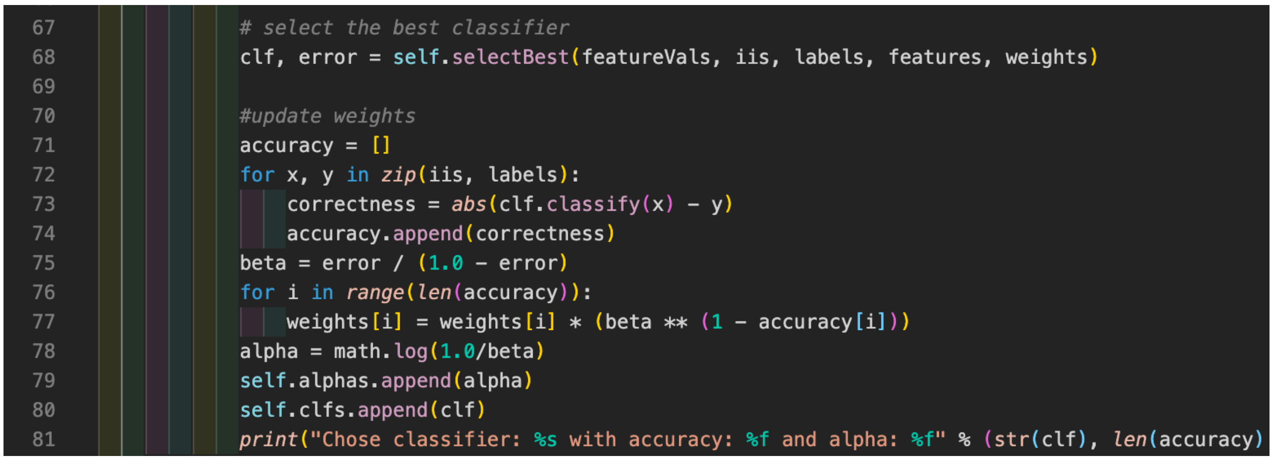
$$\epsilon_m = \frac{\sum_{n=1}^{N} w_n^{(m)} \ I(h_m(x) \neq \ t_n)}{\sum_{n=1}^{N} w_n^{(m)}}$$

[199] We use $abs(clf.classify(x) - y)$ to implement $I(h_m(x) \neq \ t_n)$ as $clf.classify(x)$ yields 1 when polarity * feature(x) < polarity * threshold.

[200~202] The errors are summed over with respect to their weights (error occurs when **I** = 1). And eventually we obtain the weighted error of this weak classifier.

** Notice that this error is different from the error above [177], that error was for finding the best threshold (obtaining a single weak classifier) for each feature. And here the error is choosing the best feature to classify the data.

(Back to the Adaboost.**train** function)

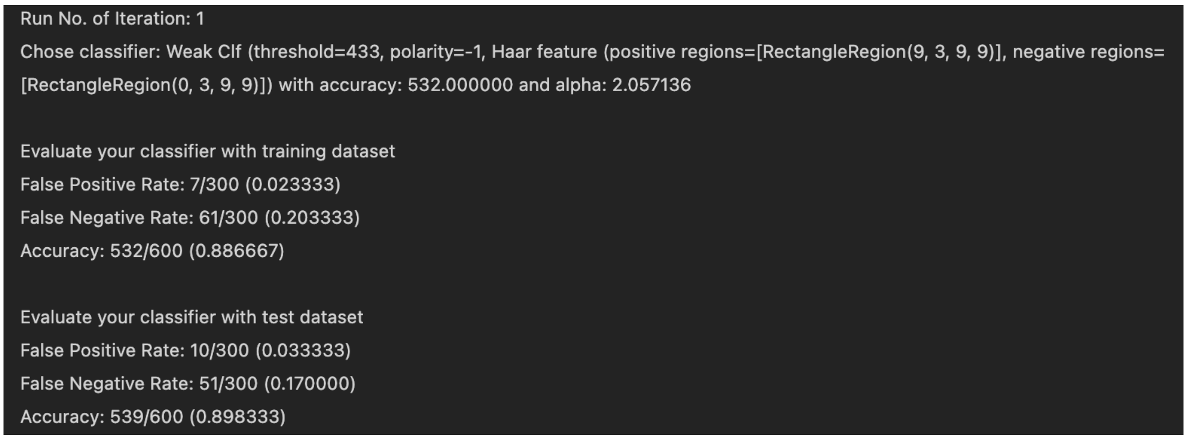


```
# select the best classifier
clf, error = self.selectBest(featureVals, iis, labels, features, weights)

#update weights
accuracy = []
for x, y in zip(iis, labels):
    correctness = abs(clf.classify(x) - y)
    accuracy.append(correctness)
beta = error / (1.0 - error)
for i in range(len(accuracy)):
    weights[i] = weights[i] * (beta ** (1 - accuracy[i]))
alpha = math.log(1.0/beta)
self.alphas.append(alpha)
self.clfs.append(clf)
print("Chose classifier: %s with accuracy: %f and alpha: %f" % (str(clf), len(accuracy)
```

Now we have the classifier with lowest error. The rest is to (1) update the weights, by intensifying the incorrectly classified and lessening the right ones; (2) calculating the weighting coefficient "alpha"; (3) adding the alpha, classifier (including best feature, best polarity and best threshold) to the strong classifier list.

The explanation of algorithm ends here.

As we run Part II and start training the function, we will get the following result.

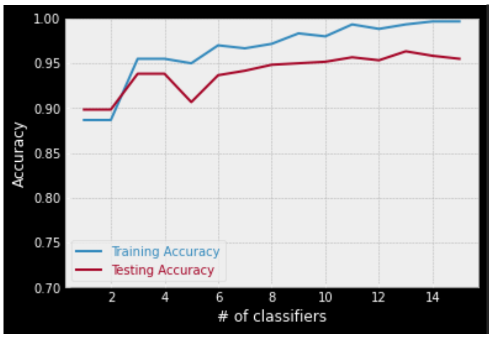


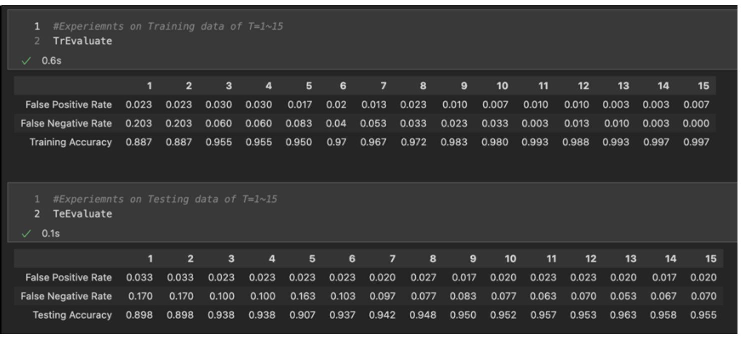
```
Run No. of Iteration: 1
Chose classifier: Weak Clf (threshold=433, polarity=-1, Haar feature (positive regions=[RectangleRegion(9, 3, 9, 9)], negative regions=
[RectangleRegion(0, 3, 9, 9)]) with accuracy: 532.000000 and alpha: 2.057136

Evaluate your classifier with training dataset
False Positive Rate: 7/300 (0.023333)
False Negative Rate: 61/300 (0.203333)
Accuracy: 532/600 (0.886667)

Evaluate your classifier with test dataset
False Positive Rate: 10/300 (0.033333)
False Negative Rate: 51/300 (0.170000)
Accuracy: 539/600 (0.898333)
```

With T set to 1 (as default), only one classifier has been selected, and its accuracy on training data is 0.8933. It correctly detects 539 out of 600 images. And I notice that there is a higher false negative rate than that of false positive rate; in other words, AdaBoost tends to confuse cars as empty parking spaces. This matches the result in the later on discussion of parking slots occupation: that AdaBoost has lower parking slots occupation than ground truth in general, but these parking spaces are actually cars.

3. Additional experiments

Test the parameter T from 1 to 15, the following tables present the results on both training data and testing data. And the decimals have been rounded to 3 places to better present.





```
1  #Experiemnts on Training data of T=1~15
2  TrEvaluate
✓ 0.6s
```

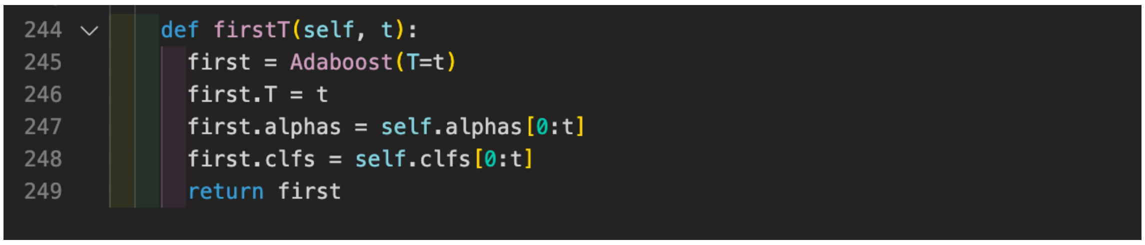
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
False Positive Rate	0.023	0.023	0.030	0.030	0.017	0.02	0.013	0.023	0.010	0.007	0.010	0.010	0.003	0.003	0.007
False Negative Rate	0.203	0.203	0.060	0.060	0.083	0.04	0.053	0.033	0.023	0.033	0.003	0.013	0.010	0.003	0.000
Training Accuracy	0.887	0.887	0.955	0.955	0.950	0.97	0.967	0.972	0.983	0.980	0.993	0.988	0.993	0.997	0.997

```
1  #Experiemnts on Testing data of T=1~15
2  TeEvaluate
✓ 0.1s
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
False Positive Rate	0.033	0.033	0.023	0.023	0.023	0.023	0.020	0.027	0.017	0.020	0.023	0.023	0.020	0.017	0.020
False Negative Rate	0.170	0.170	0.100	0.100	0.163	0.103	0.097	0.077	0.083	0.077	0.063	0.070	0.053	0.067	0.070
Testing Accuracy	0.898	0.898	0.938	0.938	0.907	0.937	0.942	0.948	0.950	0.952	0.957	0.953	0.963	0.958	0.955

The accuracy rises as T increases in general, this matches the intuition that a linear combination of weighted weak classifiers can make up a strong classifier.

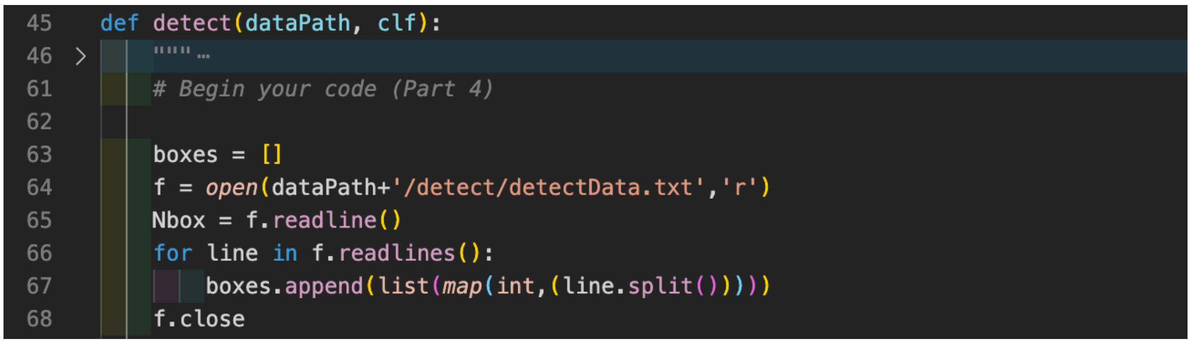
But we can see that when T >11, the testing accuracy is not any higher, it fluctuates at 0.95+. And later in the comparison, I chose T=10 as the model to compare with Yolov5.



```python
244     def firstT(self, t):
245         first = Adaboost(T=t)
246         first.T = t
247         first.alphas = self.alphas[0:t]
248         first.clfs = self.clfs[0:t]
249         return first
```

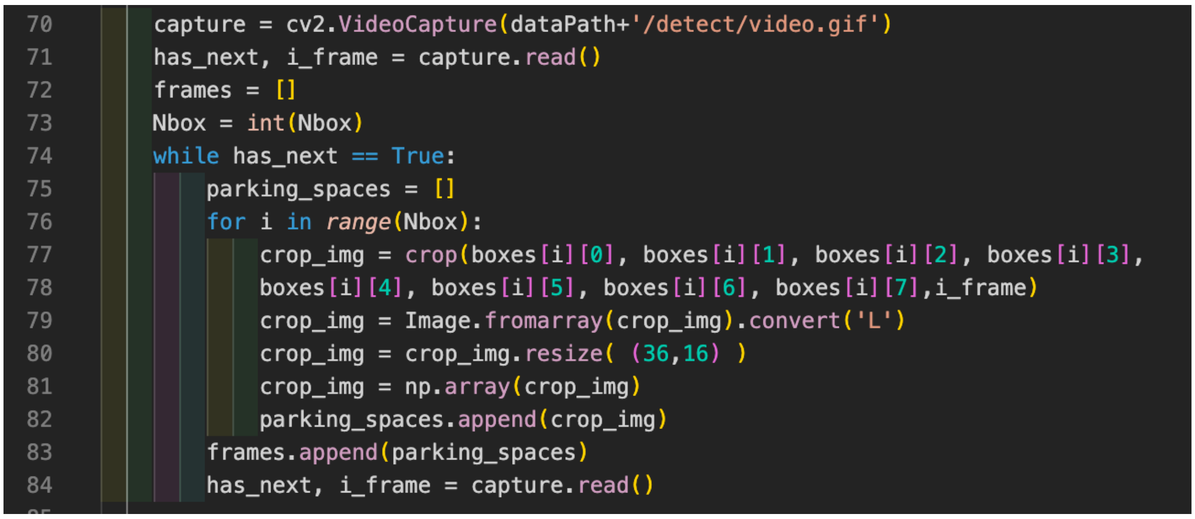
To avoid training and building features multiple times, a function “firstT” is added to the “adaboost.py” Adaboost class. It takes in the trained classifiers and return the first t weak classifiers, including their alphas and clfs.

4. Detect car



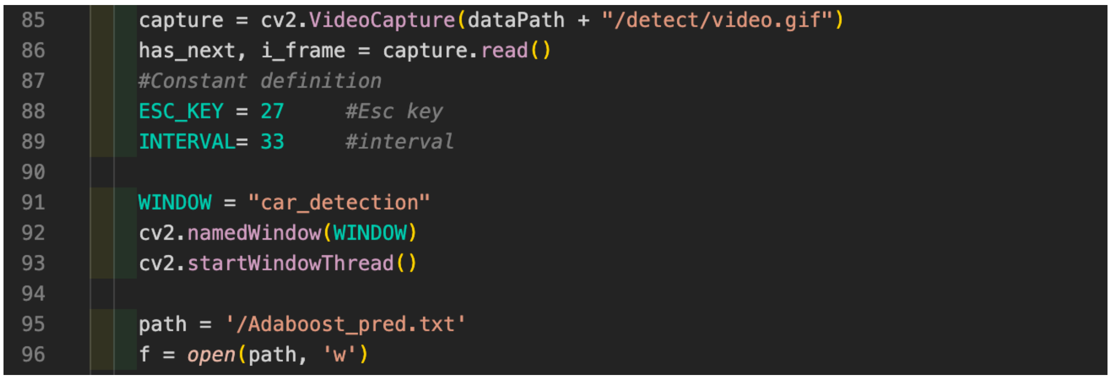
```python
45  def detect(dataPath, clf):
46      """"…
61      # Begin your code (Part 4)
62
63      boxes = []
64      f = open(dataPath+'/detect/detectData.txt','r')
65      Nbox = f.readline()
66      for line in f.readlines():
67          boxes.append(list(map(int,(line.split()))))
68      f.close
```

Read all the coordinates from txt to the list “**boxes**”.



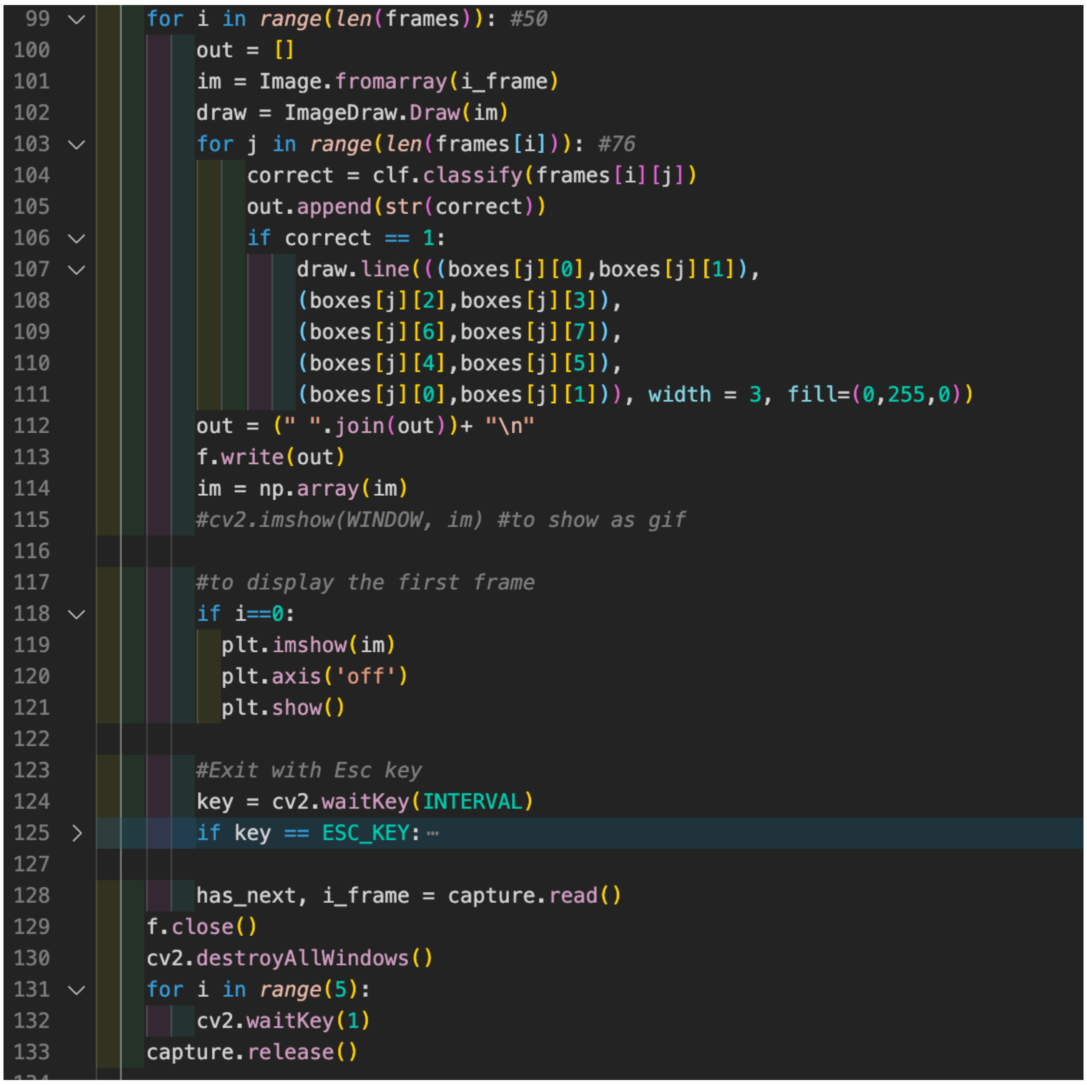
```python
70      capture = cv2.VideoCapture(dataPath+'/detect/video.gif')
71      has_next, i_frame = capture.read()
72      frames = []
73      Nbox = int(Nbox)
74      while has_next == True:
75          parking_spaces = []
76          for i in range(Nbox):
77              crop_img = crop(boxes[i][0], boxes[i][1], boxes[i][2], boxes[i][3],
78              boxes[i][4], boxes[i][5], boxes[i][6], boxes[i][7],i_frame)
79              crop_img = Image.fromarray(crop_img).convert('L')
80              crop_img = crop_img.resize( (36,16) )
81              crop_img = np.array(crop_img)
82              parking_spaces.append(crop_img)
83          frames.append(parking_spaces)
84          has_next, i_frame = capture.read()
```

[70~84] captures 76 frames from the gif and crop them to get parking space images. Then, all images are converted to grey scale, resized and added to the list “**frames**” as our input data for real detection.



```python
85      capture = cv2.VideoCapture(dataPath + "/detect/video.gif")
86      has_next, i_frame = capture.read()
87      #Constant definition
88      ESC_KEY = 27      #Esc key
89      INTERVAL= 33      #interval
90
91      WINDOW = "car_detection"
92      cv2.namedWindow(WINDOW)
93      cv2.startWindowThread()
94
95      path = '/Adaboost_pred.txt'
96      f = open(path, 'w')
```

[85~96] Initializing the parameters and file write for the later use. The prediction of AdaBoost ‘Adaboost_pred.txt’ file is saved to ‘/data/Adaboost_pred.txt’.

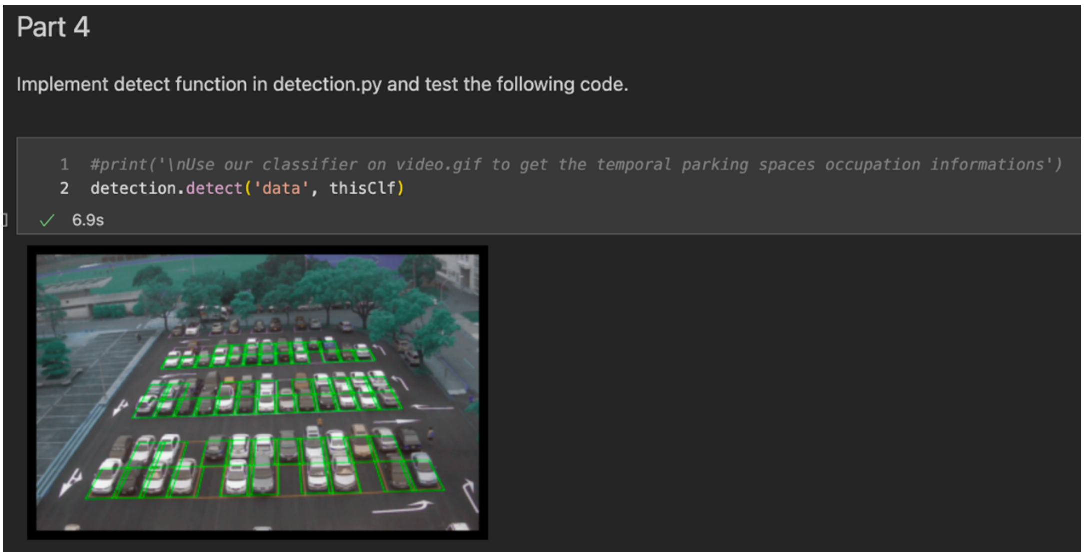


```
99  for i in range(len(frames)): #50
100     out = []
101     im = Image.fromarray(i_frame)
102     draw = ImageDraw.Draw(im)
103     for j in range(len(frames[i])): #76
104         correct = clf.classify(frames[i][j])
105         out.append(str(correct))
106         if correct == 1:
107             draw.line(((boxes[j][0],boxes[j][1]),
108             (boxes[j][2],boxes[j][3]),
109             (boxes[j][6],boxes[j][7]),
110             (boxes[j][4],boxes[j][5]),
111             (boxes[j][0],boxes[j][1])), width = 3, fill=(0,255,0))
112     out = (" ".join(out))+ "\n"
113     f.write(out)
114     im = np.array(im)
115     #cv2.imshow(WINDOW, im) #to show as gif
116 
117     #to display the first frame
118     if i==0:
119       plt.imshow(im)
120       plt.axis('off')
121       plt.show()
122 
123     #Exit with Esc key
124     key = cv2.waitKey(INTERVAL)
125     if key == ESC_KEY: ⋯
127 
128     has_next, i_frame = capture.read()
129 f.close()
130 cv2.destroyAllWindows()
131 for i in range(5):
132     cv2.waitKey(1)
133 capture.release()
```

[99~128] present double-layer For loops that go over every parking space and for each frame, in order to detect whether the space is full or not (1 or 0) with our trained classifiers. The binary results are saved to the "out" list and split by a single space for each frame. And 76 values of 1 or 0 will be written to the txt file for each frame.

[115] If we uncomment this line, a window will pop up, showing the final detection of moving gif with green boxes on it.

[118~121] plots the first frame as the follows (T=10).



Part 4

Implement detect function in detection.py and test the following code.

```
#print('\nUse our classifier on video.gif to get the temporal parking spaces occupation informations')
detection.detect('data', thisClf)
```

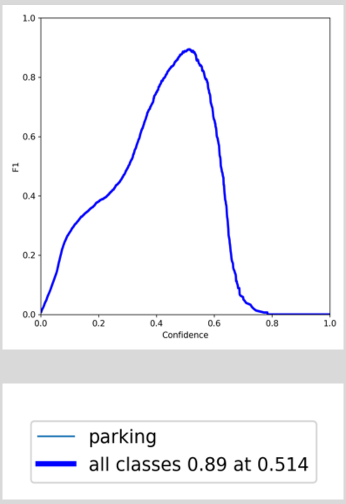
✓ 6.9s

5. Difference between AdaBoost and Yolov5

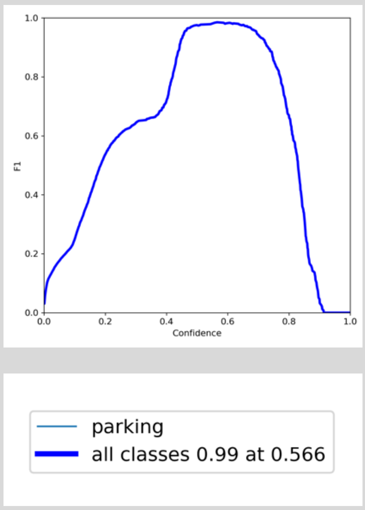
First, I tried some different hyperparameters with Yolov5, and three settings are presented below:

a. epochs 10, optimizer 'SGD' (on the sample code)
b. epochs 20, optimizer ‘Adam’
c. epochs 30, optimizer ‘AdamW’, with Cosine Annealing Learning rate

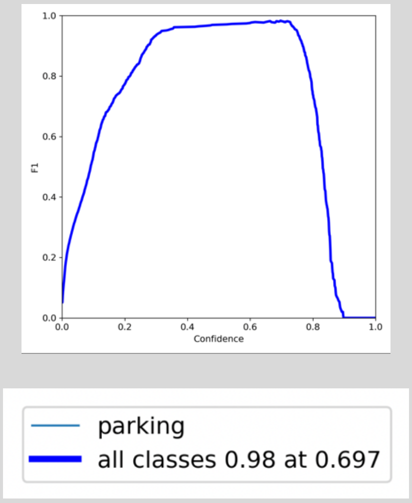
Setting a



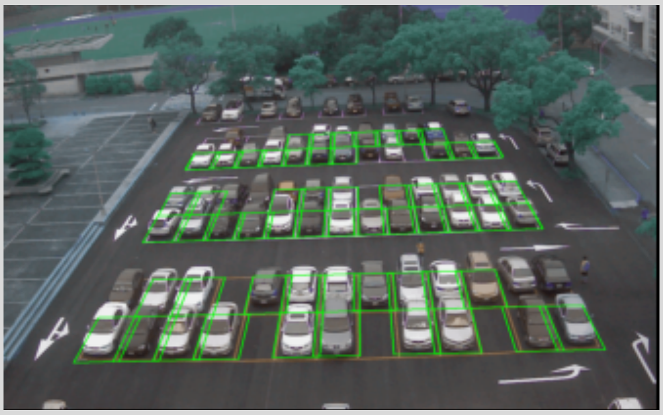
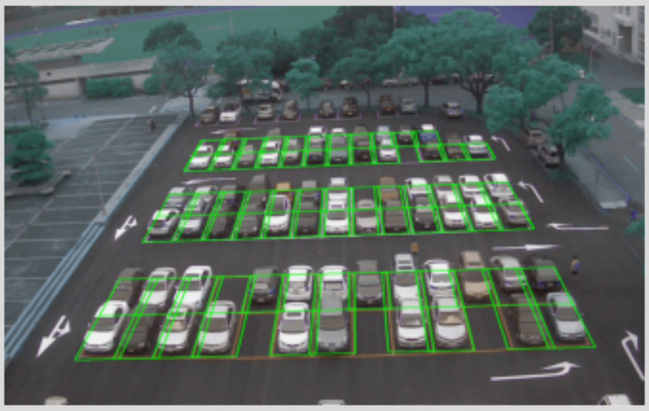
Setting b



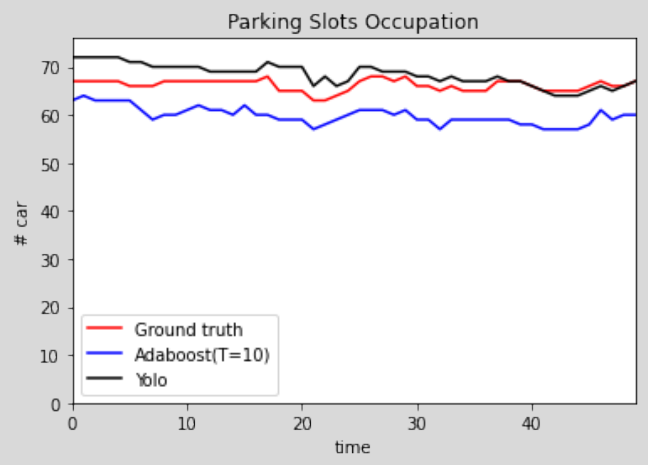
Setting c



From F1 curve we know that higher epochs and using ‘AdamW’ as optimizer is better. It appears that the first two settings present a staircase-like shape at confidence level at 0~0.5. But in setting c, the curve is smoother, being an opposite letter ‘U’, it’s probably because of the selection of optimizer. So later on, I chose “Setting c” at confidence level of 0.65 as the default model to compare with AdaBoost.

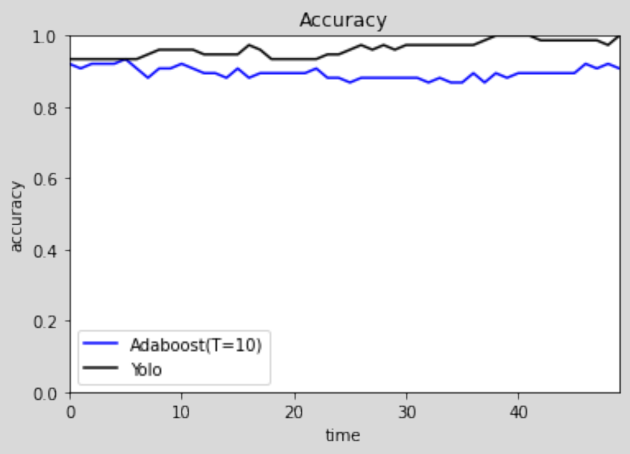
Adaboost (T=10)	**Yolov5**
	
The detection via Adaboost yields a fine result in my opinion. But the cars on the second rows(on the back) seem more likely to be misclassified. There are about 10 misclassifications as far as I can tell.	Yolov5 appears to “over” detect the cars, which makes it misclassify many empty spaces as cars. The misdetection is about 11, which yields similar accuracy as AdaBoost for the first frame.

Parking Slots Occupation



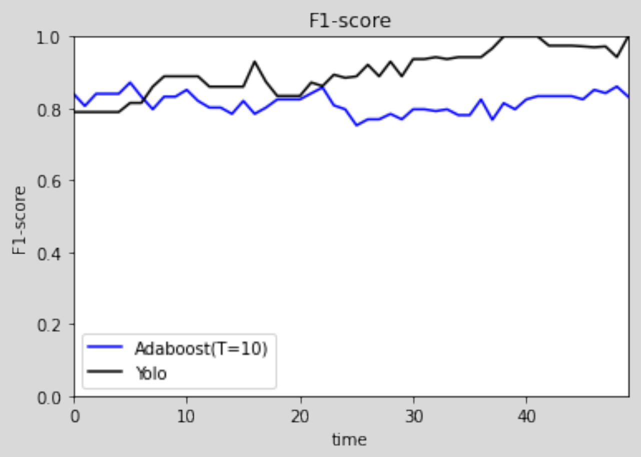
Yolov5 seems to have a higher parking slots occupation in general, and adaboost have lower. But their difference from the Ground truth are very close.

Accuracy



Both models achieve high accuracy. But in general, Yolov5 still performs better than AdaBoost as the graph has shown. And during time frame 38~40, Yolo can even achieve 100% accuracy.

F1-score



$$F1 - score: tp/(tp + \ 1/2\ (fp + fn))$$

Since accuracy cannot take how data is distributed into account, (in our cases, there are many occupied parking spaces, but very few empty spaces), we would like to see the F1-score. F1-score is preferred when the False Negatives and False Positives are crucial.

We can see that in the first few frames, such as at time frame 1~5, AdaBoost has a slightly higher f1-score than Yolo.

Section II

Problems I’ve encountered during the implementation

1. I found it quite hard to understand the codes of Adaboost at first, even after watching Professor Li’s introducing video. So, I did some more research and read the ppts from others’ lectures. And it turned out that I got stuck on the filters Viola/Jones and related calculations. I misinterpreted the connections between the algorithm itself and the Viola/Jones Face Detection part.
2. Since there is also a function called “crop()” from the package Pillow, I didn’t think much of it and apply this function on the data. And the processed image was not the required size, and it appeared to be a little skewed comparing to the one on the spec. I was typing my codes on main.ipynb at first, so I didn’t notice that there were a crop function in “detection.py”… The lesson learnt was that I really need to scan all the given files before my implementation.
3. Part III requires to test the parameters T for Adaboost. A For loop running for 10 times first came to my mind, but this is really inefficient considering the training function will do the calculations for features and iterations multiple times. Therefore, I added a function “firstT” to the Adaboost class that takes in a parameter t, and returns the first t alphas and clfs as a new classifier. But I think this method is yet to be improved.
4. Yolov5 modified Api, leading to an error. Thanks to TA and classmates, you mentioned and solved this problem on the discussion section, this was really helpful.

Reference

- https://courses.cs.washington.edu/courses/cse455/16wi/notes/15_FaceDetection.pdf
- https://www.vision.rwth-aachen.de/media/course/WS/2017/machine-learning/ml17-part10-adaboost.pdf
- https://www.vocal.com/video/face-detection-using-viola-jones-algorithm/