

CP600 Image Processing - Lane Detection Report

Brian Tiner

Millions of people lose their lives to traffic accidents every year. In order to combat this, advanced driving systems have been researched very heavily in the last decade or so. Based on the research that I have conducted, I attempted to design a robust lane detection system that would be able to detect lanes using images from the point of view of the car. The system employs several different preprocessing techniques in order to produce a clean image used for lane detection. The system also provides three different edge detection algorithms so the accuracy of each can be compared. The edge detection algorithms that have been implemented are Canny edge detection, laplacian edge detection, and a pre-trained neural network. Multiple different tests are run using each algorithm and the accuracy of lane detection is recorded to see which system works best in different scenarios. First, each stage of the lane detection system will be described in the order that they are executed.

The system begins by reading each image from a group of jpg files and assigning them to an array of images. Then, the first preprocessing is executed on each image. The images are thresholded to binary images so lanes are more defined from the road for easier edge detection. This thresholding is done by examining the pixel values in each image. Pixel values range from 0-255 where 0 is black and 255 is white. If the value of the pixel is above the threshold, then the pixel is made white because it is considered a lane. If the value of the pixel is below the threshold, then the pixel is made black because it is not considered part of a lane. This method works well to define lanes from the road but when shadows are introduced, it can affect the accuracy because lanes are darker than usual. After thresholding is done, one of the edge detection algorithms can be selected to extract the edges from the images. The edge detection algorithms will be described in the next section. After the edges have been extracted from the images, more preprocessing is done to prepare the images for lane detection. The images are

then cropped to a region of interest. The region of interest is a triangle where the three vertices of the triangle are the two bottom corners of the image and the center of the image. This ideally crops the images to where only the road in front of the vehicle can be seen and no outside noise will be considered in the lane detection. The region of interest is cropped by manipulating the numpy array that represents the image and making every pixel outside the cropped area black. After the region of interest is cropped, the images are converted to grayscale because some of the edge detection algorithms produce images with colour channels rather than grayscale. The images are then split down the middle into two separate images. This is done so the left lane and the right lane surrounding the car can be detected separately. After the images are split, a Hough transform is applied to each side to detect the left and right lanes separately. The Hough transform identifies a perpendicular line from the origin of the image to the line that has been detected. It returns the length and the angle of the perpendicular line that has been identified in the image. Using the length and angle of the perpendicular line, the X and Y coordinates of the two ends of the line can be calculated. Using the two X and Y coordinates, the lines are drawn on the original image so you can see if the lanes are detected accurately. The accuracy is also calculated using the X and Y coordinates by checking if the coordinates of the proper lane are within the coordinates calculated. Each stage of the system saves the images to directories so each individual preprocessing method can be examined to see if it is producing correct results.

There are three different edge detection algorithms that are implemented for this lane detection system. I implemented three different algorithms so the accuracy can be compared and the best algorithm can be identified. As mentioned before, the three algorithms are Canny edge detection, laplacian edge detection, and a pre-trained neural network named Holistically nested edge detection. Each of the algorithms will be described in detail.

The Canny edge detection algorithm is a popular edge detection method in computer vision applications. It approaches edge detection by using image thresholding. The algorithm defines an upper and a lower threshold for pixel values. It then examines every pixel in a given

image. The thresholds are used to decide if a pixel is part of an edge or not. If the pixel value is above the upper threshold, then it is considered an edge and the pixel is made white. If the pixel value is below the lower threshold, then it is not considered an edge and the pixel is made black. If the pixel value is between the two thresholds, then the pixel is only considered an edge if it is connected to another pixel that is already considered an edge. While somewhat simple, the Canny edge detector produces favourable results by detecting the lane edges from the road.

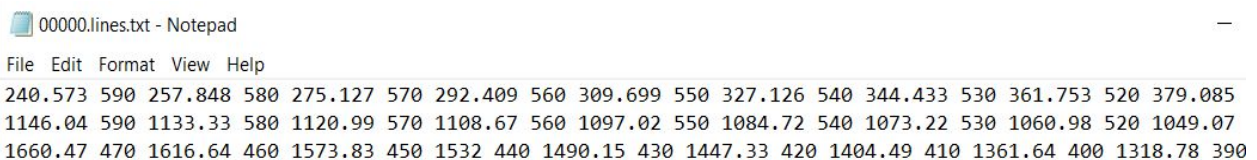
The laplacian edge detection is one of many convolution filters used in computer vision. It specifically is used for edge detection. It uses a three by three kernel matrix where the center value of the matrix is negative eight and the other eight values are one. The matrix is used to execute matrix multiplication on the pixel values and ideally identify edges. The kernel matrix is run over every pixel in the image. The results of the matrix multiplication is then assigned to the pixel that is being examined. It can be used to identify edges that are horizontal, vertical, or diagonal.

The final edge detection algorithm implemented is the pre-trained Holistically nested neural network. It is pre-trained for edge detection and provided by the python library OpenCV. The trained model was downloaded to my system and loaded into the program for use. I first needed to implement a CropLayer class for the neural network to detect edges. The CropLayer class contains the functions getMemoryShapes and forward. The getMemoryShapes function is used to identify the dimensions of the input image to the network. It returns the height, width, colour channels, and batch size. The forward method is responsible for actually cropping the image in the forward pass of the neural network. The Holistically nested edge detection works in several stages for each image. It conducts several convolutions on the image where each convolution is followed by average pooling. It has five convolution layers and five average pooling layers. Each convolution layer works similar to the laplacian edge detection where it runs a kernel matrix over the image and produces some defined edges. After each convolution, average pooling is done to reduce the dimensionality and summarize the output of the

convolution layers. After the image is run through the network, it is resized to the original image dimensions. The pre-trained neural network yields very good edges and preserves object boundaries well. However, it takes longer to run than simple algorithms.

Each of these algorithms are applicable to lane detection systems. Lane detection is an image processing problem that needs to be able to differentiate between lanes and the road. The most robust way to achieve lane detection is through detecting the edges of the lanes from the road they are painted on. As long as there are actual differences between the lanes and road in images, edge detection algorithms can run successfully. Shadows and other noise cause problems in edge detection by making the lanes less defined from the road. The most robust edge detection algorithms, such as the Holistically nested neural network, can identify lanes even in worst case scenarios. This makes edge detection essential for a lane detection system to function correctly.

The dataset used for the lane detection system implemented was provided by the Chinese University of Hong Kong. <https://xingangpan.github.io/projects/CULane.html> It contains over 100,000 images of the road from a vehicles perspective on the streets and highways of Beijing. The data is distributed among many files where each file contains about 180 images. Each image also has an associated text file that has information about the lanes in the image for accuracy calculations. The text files provided for accuracy calculations are difficult to deal with for a few reasons. It provides several X and Y coordinates for each lane in the image as seen in



```
240.573 590 257.848 580 275.127 570 292.409 560 309.699 550 327.126 540 344.433 530 361.753 520 379.085
1146.04 590 1133.33 580 1120.99 570 1108.67 560 1097.02 550 1084.72 540 1073.22 530 1060.98 520 1049.07
1660.47 470 1616.64 460 1573.83 450 1532 440 1490.15 430 1447.33 420 1404.49 410 1361.64 400 1318.78 390
```

the image here. For this project, I only attempted to identify the lanes on either side of the car rather than the lanes that are over a lane or on the other side of the road. Each series of pictures has varying lines in their respective text file because sometimes the vehicle is in a single lane street and sometimes it is in different lanes on a multilane highway. Because of this,

the text files were difficult to use arbitrarily for every test case. For each test I ran on the images, I only used a single file of 180 images. By doing this, I could preprocess the text files to remove the lanes that I did not want to identify for each test separately. It is also difficult to accurately identify if the coordinates of each text file is the lane that was recognized by the Hough transform. For solid lanes, the Hough transform would find a longer line and have a wider spread of X and Y values. Checking if the coordinates of the text file were within a larger bound worked well. When there were dotted lanes, the Hough transform would identify a shorter line because only one or two dots in the lane were visible. The X and Y coordinates would be closer together but on the correct slope of the lane. While it would still plot the correct lane on the image, the accuracy calculation would not identify that all the coordinates were between the values returned by the Hough transform. So the images provided by the dataset would be labeled correctly but the accuracy calculation did not represent this well.

The entire system was implemented in python 3.8.3 using Jupyter notebook. The packages used were OpenCV and numpy. Numpy is used because the images are represented through Numpy arrays. Numpy was used to manipulate the image for some preprocessing such as cropping a region of interest. The OpenCV package was used for the majority of the preprocessing methods and all of the edge detection algorithms. The Holistically nested neural network was provided through OpenCV as well. The entire system was executed on my laptop which uses Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz and 12 GB of RAM.

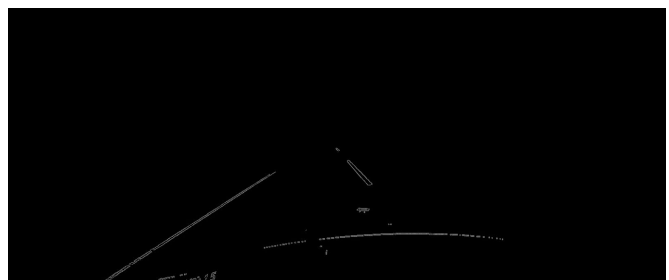
There are very many different parameters that were tuned for the lane detection system to work correctly. Many of the preprocessing methods are very reliant on certain values such as thresholds or minimum vote counts. For the initial image thresholding, different threshold values were used to define the lanes from the road. Initially I used the midpoint of pixel values which is 127. As seen in the image here is thresholding using the value 127 as



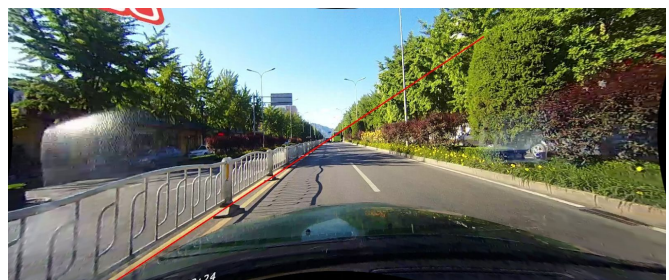
a threshold. Within the lane, there are some pixels that are not made black and can have some influence on the edge detection. I then changed the threshold value to 150 to try to account for noise in the image. As you can see in the second



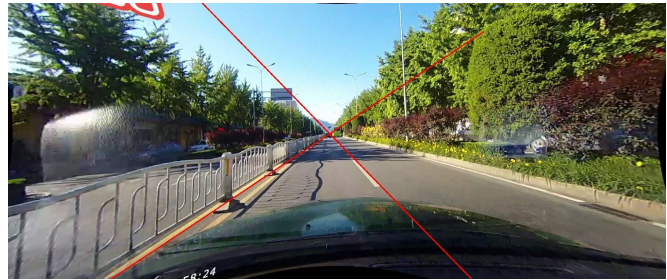
image here, the pixel values within the lane are all black and it is much easier to detect the edges. Cropping the region of interest also had some tuning to find the ideal region of interest for the lane detection. Initially, the top vertex of the triangle that represents the region of interest was put slightly above the center of the image. I wanted to get a full view of the lane in front of the vehicle to get the most accuracy edge detection. However, as you can see in the image here it introduced other edges within the region of interest which hurts the lane detection accuracy. After this, I decided to make the region of interest in the dead center of the image. In the second image here, you can see that while the lanes are still visible. There is no outside noise that influences the edge detection. The final



parameter that was tuned is the minimum vote count that the Hough transform needs to define a line. Initially, the vote count was set to 100 in an attempt to only identify lanes and not other edges within the images. While this would still work with most solid lanes in good scenarios, it proved difficult to identify dotted lanes or less defined lanes. In the



first image here, only the solid lane is detected with a minimum vote count of 100. To account for this, I changed the minimum vote count to a much lower number. With a minimum vote count of 10, the solid lanes were still detected just as well while also



being able to detect dotted lanes better. The second image here shows both lanes being detected with a minimum vote count of 10. Each of these parameters influence the final product of lane detection and if not set properly count cause issues in the lane detection accuracy.

The results of the lane detection system must be examined to compare the accuracy of the different edge detection algorithms used. As mentioned before, each test was executed on a small sample of the dataset so I could preprocess the text files properly to identify only the two lanes on either side of the car. The accuracy for the left lane and the right lane were calculated separately because of the difficulty in calculating accuracy. For the first two tests, ideal conditions were used where the car is driving on a highway in the left and middle lanes. For the third, fourth, and fifth tests, less ideal conditions were used where the car was driving on city streets with shadows and noise. For the final two tests, very bad conditions were used where the car was in heavy traffic on busy city streets. Before I begin to analyze the results, the low accuracy for the right lane must be considered. Like mentioned before, it was very difficult to calculate the accuracy for the right side of the vehicle. The way the accuracy was calculated was by checking if every point provided in the text file was within the bounds of the resulting Hough transform. When the Hough transform tries to identify dotted lanes, it recognizes a shorter line. Since accuracy is calculated by checking if every point is between the bounds of the Hough transform, it does not recognize that every point is between those bounds. While solid lanes are detected better, the dotted lanes are still detected some of the time.

Method	Test 1 Left	Test 1 Right	Test 2 Left	Test 2 Right	Test 3 Left	Test 3 Right	Test 4 Left	Test 4 Right	Test 5 Right	Test 5 Left
Canny	0.927	0.000	0.938	0.011	0.584	0.039	0.789	0.029	0.793	0.054
Laplacian	0.933	0.000	0.944	0.000	0.640	0.045	0.883	0.017	0.830	0.060
HED Neural Network	0.972	0.017	0.966	0.017	0.747	0.039	0.918	0.089	0.878	0.073

Method	Test 6 Left	Test 6 Right	Test 7 Left	Test 7 Right
Canny	0.025	0.000	0.050	0.000
Laplacian	0.174	0.000	0.066	0.000
HED Neural Network	0.451	0.116	0.250	0.000

Now given the results in the two tables above, there are several observations that can be made. In the first two tests the left side is detected with a high degree of accuracy. The Holistically nested neural network has the most accurate results out of the three algorithms. The right side is not calculated very accurately but this is because the right side of the car is primarily dotted lanes for each of these images. For the third, fourth, and fifth tests, the neural network also provides the best results. Since there are more shadows and noise in the image, each of the algorithms produce less accurate results than the first two tests. The right side does have slightly higher accuracy and this is because some of the time there is a solid lane on the right side of the car in these tests. In the final two tests, the accuracy of every algorithm is very low. This is because the lanes are much harder to detect with heavy traffic. While the Canny and Laplacian edge detection can barely identify any lanes, the neural network can still identify a reasonable amount given the difficult scenario. This shows that the Holistically nested neural network is the most robust algorithm out of the three implemented.

To compare my results to other work, I found lane detection research that uses the same dataset. <https://arxiv.org/pdf/1905.03704.pdf> Provided in the table here are various algorithms

Category	E ⁺ Net	ResNet-18	VGG-16 ⁺	ReNet ⁺	DenseCRF ⁺	MRFNet ⁺	ResNet-50 ⁺	ResNet-101 ⁺	SCNN ⁺
Normal	88.4	89.8	83.1	83.3	81.3	86.3	87.4	90.2	90.6
Crowded	67.0	68.1	61.0	60.5	58.8	65.2	64.1	68.2	69.7
Night	61.4	64.2	56.9	56.3	54.2	61.3	60.6	65.9	66.1
No line	42.9	42.5	34.0	34.5	31.9	37.2	38.1	41.7	43.4
Shadow	63.4	67.5	54.7	55.0	56.3	59.3	60.7	64.6	66.9
Arrow	81.9	83.9	74.0	74.1	71.2	76.9	79.0	84.0	84.1
Dazzle light	57.4	59.8	49.9	48.2	46.2	53.7	54.1	59.8	58.5
Curve	62.6	65.5	61.0	59.9	57.8	62.3	59.8	65.5	64.4
Crossroad	2768	1995	2060	2296	2253	1837	2505	2183	1990
Total	68.8	70.5	63.2	62.9	61.0	67.0	66.7	70.8	71.6

that were tested on the same dataset. They tested a much larger sample of the dataset than I did but results can still be compared. In good conditions, many of my algorithms have higher accuracies for the left lane. However, if you combine my accuracy for the right lane it would drop lower than their accuracy. Regardless, the algorithms I have implemented are on par with the accuracies reported here for normal conditions. Given more difficult conditions, the algorithms implemented in this research detect lanes more accurately than mine. Given many different scenarios, they have usually above 60% accuracy. The best algorithm that I implemented in terrible conditions cannot yield more than about 45% accuracy. While this is better than my tests, the Holistically nested neural network can still compete with the results published here. If the right lane accuracy was calculated properly for my implementation, the accuracies of the Holistically nested neural network would be displayed similar to the accuracies recorded here. The Canny and Laplacian edge detection cannot hold up with more complex algorithms.

The code of my implementation is provided below. It is saved as a ipynb file which is used in Jupyter notebook. It should be run incrementally in the order provided. The only user input to the program is to select which edge detection algorithm to execute. Each step of the system is output to different files so you can see every preprocessing step as well as the edge detection. The accuracy is output at the end of the program and the final lanes with line plotted are written to files. To run the code there are several files that need to be included in my submission. The images are loaded from a folder into the program. There is a file named list.txt

which contains the names of the image files to be loaded into the program. The

hed-edge-detector folder must be included as well to load the pre-trained neural network.

```
#imports
```

```
import cv2
import numpy as np
from decimal import Decimal
```

```
#getting image names
f = open("list.txt")
#setting list of images and line files
i = 0
original_imgs = []
line_files = []
line_lists = []
for line in f:
    temp = []
    original_imgs.append(cv2.imread(("05151640_0419.MP4/" + line.strip()), 1))
    #preprocessing correct value files
    line_files.append(open("05151640_0419.MP4/" + line.strip()[0:5] + ".lines.txt"))
    l = line_files[i].readline().strip()
    while l != "":
        temp.append(l)
        l = line_files[i].readline().strip()
    if len(temp) == 3:
        temp.pop(0)
    elif len(temp) == 4:
        temp.pop(0)
        temp.pop()
    line_lists.append(temp)
    cv2.imwrite("images/img" + str(i) + ".jpg", original_imgs[i])
    i+=1
```

```
imgs = original_imgs.copy()
f.close()
```

```
#Helpers
```

```
#region of interest function
def roi(img, vertices):
    #creating mask
    mask = np.zeros_like(img)
    #setting colour to grayscale
    colour = (128,128,128)
    #filling mask
    cv2.fillPoly(mask, vertices, colour)
    #returning cropped image
    return cv2.bitwise_and(img, mask)
```

#CropLayer for Neural Network

```
class CropLayer(object):
```

```
    def __init__(self, params, blobs):
```

```
        self.xstart = 0
```

```
        self.xend = 0
```

```
        self.ystart = 0
```

```
        self.yend = 0
```

```
    def getMemoryShapes(self, inputs):
```

```
        inputShape, targetShape = inputs[0], inputs[1]
```

```
        batchSize, numChannels = inputShape[0], inputShape[1]
```

```
        height, width = targetShape[2], targetShape[3]
```

```
        self.ystart = int((inputShape[2] - targetShape[2]) / 2)
```

```
        self.xstart = int((inputShape[3] - targetShape[3]) / 2)
```

```
        self.yend = self.ystart + height
```

```
        self.xend = self.xstart + width
```

```
        return [[batchSize, numChannels, height, width]]
```

```
    def forward(self, inputs):
```

```
        return [inputs[0][:,self.ystart:self.yend,self.xstart:self.xend]]
```

#Threshold image

```
i = 0
```

```
for img in imgs:
```

```
    (thresh, imgs[i]) = cv2.threshold(img, 150, 255, cv2.THRESH_BINARY)
```

```
    cv2.imwrite("threshold/threshold" + str(i) + ".jpg", imgs[i])
```

```
    i+=1
```

#canny edge detection

```
method = int(input("Canny(1), Neural Network(2), Laplacian(3): "))
```

```
if method == 1:
```

```
    i = 0
```

```
    for img in imgs:
```

```
        imgs[i] = cv2.Canny(img,100,150)
```

```
        cv2.imwrite("edges/edges" + str(i) + ".jpg", imgs[i])
```

```
        i+=1
```

#Neural Network

```
elif method == 2:
```

```
    #loading trained neural network
```

```
    network = cv2.dnn.readNetFromCaffe("hed-edge-detector/deploy.prototxt",
```

```
"hed-edge-detector/hed_pretrained_bsds.caffemodel")
```

```
    cv2.dnn_registerLayer('Crop', CropLayer)
```

```
    i = 0
```

```
    for img in imgs:
```

```
        print("Iteration: " + str(i))
```

```
        #getting binary large object from image
```

```
        blob = cv2.dnn.blobFromImage(img, scalefactor=1.0, size=(1640, 590),
```

```
mean=(104.00698793, 116.66876762, 122.67891434), swapRB=False, crop=False)
```

```

#setting input
network.setInput(blob)
#getting output
output = network.forward()
#manipulating output back to image
output = output[0, 0]
output = cv2.resize(output, (img.shape[1], img.shape[0]))
output = 255 * output
imgs[i] = output.astype(np.uint8)
cv2.imwrite("edges/edges" + str(i) + ".jpg", imgs[i])
i+=1

```

elif method == 3:

```

i = 0
for img in imgs:
    output = cv2.Laplacian(img,cv2.CV_64F)
    output = cv2.resize(output, (img.shape[1], img.shape[0]))
    output = 255 * output
    imgs[i] = output.astype(np.uint8)
    cv2.imwrite("edges/edges" + str(i) + ".jpg", imgs[i])
    i+=1

```

#cropping region of interest

```

i = 0
for img in imgs:
    if len(img.shape) == 2:
        h, w = img.shape
    else:
        h, w, c = img.shape
    #Cropping to triangle shape
    vertices = [(0, h), (w / 2, h / 2), (w, h)]
    imgs[i] = roi(img, np.array([vertices], np.int32))
    cv2.imwrite("roi/roi" + str(i) + ".jpg", imgs[i])
    i+=1

```

#Hough Transform

```

temp_imgs = original_imgs.copy()
start_row, start_col = int(0), int(0)
end_row, end_col = int(h), int(w * .5)
i = 0
lcount = 0
rcount = 0
ltotal = 0
rtotal = 0
for img in imgs:
    if len(img.shape) == 3:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    #Splitting images
    imgleft = img[start_row:end_row , start_col:end_col]
    imgright = img[start_row:end_row , end_col:w]
    #getting left and right lane

```

```

linesleft = cv2.HoughLines(imgleft,1,np.pi/170,10)
linesright = cv2.HoughLines(imgright,1,np.pi/170,10)
#writing left lane on image
if linesleft is not None:
    for rho,theta in linesleft[0]:
        count = 0
        total = 0
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a*rho
        y0 = b*rho
        x1 = int(x0 + 1000*(-b))
        y1 = int(y0 + 1000*(a))
        x2 = int(x0 - 1000*(-b))
        y2 = int(y0 - 1000*(a))
        if abs(y1 - y2) > 300:
            cv2.line(temp_imgs[i],(x1,y1),(x2,y2),(0,0,255),2)
        if len(line_lists[i]) > 0:
            line = line_lists[i][0].split(" ")
            for j in range(0, len(line) - 1, 2):
                if Decimal(line[j]) > x1 and Decimal(line[j]) < x2 and Decimal(line[j + 1]) < y1 and
Decimal(line[j + 1]) > y2: count += 1
                total += 1
            if count/total >= 0.5: lcount+=1
            ltotal+=1
#writing right lane on image
if linesright is not None:
    for rho,theta in linesright[0]:
        count = 0
        total = 0
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a*rho
        y0 = b*rho
        x1 = int(x0 + 1000*(-b))
        y1 = int(y0 + 1000*(a))
        x2 = int(x0 - 1000*(-b))
        y2 = int(y0 - 1000*(a))
        if abs(y1 - y2) > 300:
            cv2.line(temp_imgs[i],(x1 + 820,y1),(x2 + 820,y2),(0,0,255),2)
        if len(line_lists[i]) > 1:
            line = line_lists[i][1].split(" ")
            for j in range(0, len(line) - 1, 2):
                if Decimal(line[j]) > x1 + 820 and Decimal(line[j]) < x2 + 820 and Decimal(line[j + 1])
< y1 and Decimal(line[j + 1]) > y2: count += 1
                total += 1
            if count/total >= 0.5: rcount+=1
            rtotal+=1
    cv2.imwrite("lines/houghlines" + str(i) + ".jpg",temp_imgs[i])
    i += 1
print("Left Accuracy: " + str(lcount/ltotal))

```

```
print("Right Accuracy: " + str(rcount/rtotal))
```