COMP6200 Machine Learning

Homework 4

Name: Ibna Kowsar
Affiliation: MSCS
Submission Date: 02/25/2024

Table of contents

Introduction:

In this report, we focus on two different experimental tasks. For the first task, we used pictures of 10 individual people from different age groups. From the picture of these people named image1, image2,....image10 we extract the facial portion using the Haar-cascade classifier and save the face images in a directory called faces. Later, we take a random facial image and match it with all other images by using BF-matcher from OpenCV. For the second task, I used my cellphone to take a picture of a parking space and then detected a line using Canny edge detection and Hugh transform and showed available parking space. This report illustrates both tasks in detail in the following sections.

Implementation:

Task 01:

1.1

Figure 1: Images of 10 different people from different age groups

For this task I created a class called FaceMatcher which will have all the methods and instance variable initialized in it. For instance, I have **load_image()** [shown in Figure 02] method to all of these images as shown in Figure 01.

```
def load_image(self, file_path):
    image = cv2.imread(str(file_path))
    if image is None:
        print(f"Image {file_path} not found or unable to load.")
        return None
    return image
```

Figure 2: Loading image from the given file_path

Once I load the images, now according to the question I will have to extract facial images and save it to a directory. Therefore, I initialize all the global variables in the contractor (i.e., root data path, save_path of facial images, cascade classifier initialization, and other necessary variables) as illustrated in Figure 03.

```
import cv2
from pathlib import Path
from random import choice

class FaceMatcher:
    def __init__(self, data_path):
        self.data_path = Path(data_path)
        self.faces_path = self.data_path / "faces"
        self.faces_path.mkdir(exist_ok=True)
        self.face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
        self.orb = cv2.ORB_create()
        self.bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
```

Figure 03: FaceMatcher class and contractor for Task 01

After that, I have 2 methods to detect faces from images and save them to a directory. Here is a code snipped in Figure04 that uses a haar cascade classifier for face detection and saves face images to a directory.

```
def detect_faces(self, image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    faces = self.face_cascade.detectMultiScale(gray, 1.1, 4)
    return [image[y:y+h, x:x+w] for (x, y, w, h) in faces]

def save_faces(self, faces, image_number):
    face_paths = []
    for idx, face in enumerate(faces):
        face_filename = f"face{image_number}_{idx}.png"
        face_path = self.faces_path / face_filename
        cv2.imwrite(str(face_path), face)
        face_paths.append(face_path)
    return face_paths
```

Figure 4: Detecting face from the image and saving in a directory

Figure05 shows output images that were saved after detecting face.

Figure 05: output facial images from haar-cascade classifier

After that, I use **match_faces()** method to match a random image with the extracted facial images which use BF matcher from OpenCV and ORB detector from OpenCV as shown in Figure 06.

```
def find_keypoints_descriptors(self, image):
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    keypoints, descriptors = self.orb.detectAndCompute(gray_image, None)
    return keypoints, descriptors

def match_faces(self, descriptors_to_match, image_paths):
    best_match_path = None
    best_match_score = -1
    for path in image_paths:
        image = self.load_image(path)
        if image is not None:
            keypoints, descriptors = self.find_keypoints_descriptors(image)
            if descriptors is not None and len(descriptors) > 0:
                matches = self.bf.match(descriptors_to_match, descriptors)
                matches = sorted(matches, key=lambda x: x.distance)
                match_score = len(matches)
                if match_score > best_match_score:
                    best_match_score = match_score
                    best_match_path = path
    return best_match_path, best_match_score
```

Figure 06: Face match with a random image using BF_matcher and ORB

Here is my driver code (Figure 07) that creates an instance of the FaceMatcher class and calls load_image, detect_faces, save_faces, match_faces, etc.

```python
# Defining root dir
data_path = "/root"
# Creating FaceMatcher object to initialize all the required methods
face_matcher = FaceMatcher(data_path)

# Detecting and saving faces from images
face_images_paths = []
for i in range(1, 11):
    image_path = face_matcher.data_path / f"image{i}.png"
    if not image_path.is_file():
        image_path = face_matcher.data_path / f"image{i}.jpg"
    image = face_matcher.load_image(image_path)
    if image is not None:
        faces = face_matcher.detect_faces(image)
        face_paths = face_matcher.save_faces(faces, i)
        face_images_paths.extend(face_paths)

# Selecting a random face and finding its descriptions
random_face_path = choice(face_images_paths)
random_face_image = face_matcher.load_image(random_face_path)
_, descriptors_to_match = face_matcher.find_keypoints_descriptors(random_face_image)

# Matching the face descriptors against all original images
original_image_paths = [str(face_matcher.data_path / f"image{i}.png") if (face_matcher.data_path / f"image{i}.png")
                        str(face_matcher.data_path / f"image{i}.jpg") if (face_matcher.data_path / f"image{i}.jpg")
                        for i in range(1, 11)]
original_image_paths = [path for path in original_image_paths if path is not None]
best_match_path, best_match_score = face_matcher.match_faces(descriptors_to_match, original_image_paths)

print(random_face_path, best_match_path, best_match_score)

```

Figure 07: Driver code for task 01 to load the image, detect face, and match with a random image

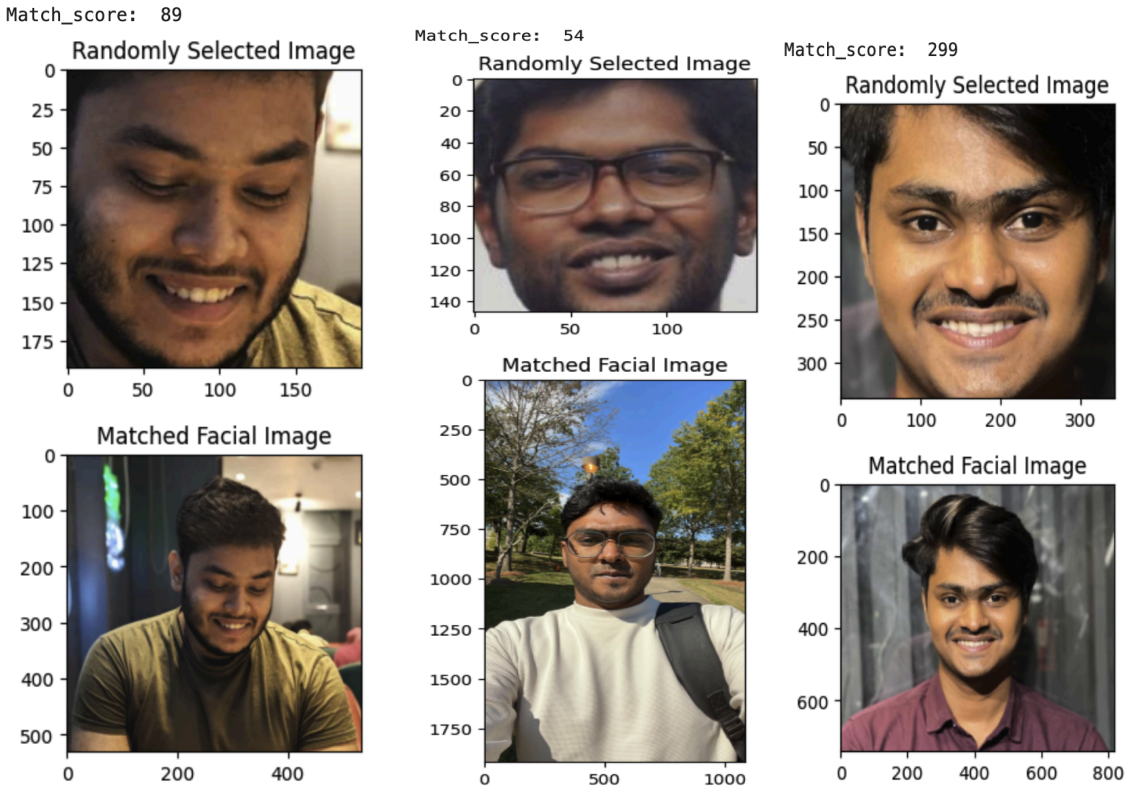

Figure 08: Sample of the randomly selected image and matched output with Brute-Force match Score

I tested this algorithm **over 30 times** and found that in one or a few scenarios it is giving the wrong prediction as shown in Figure 08 second picture.

Task02:

For this task, I took an image of a parking space with white lines in between each parking space.

Firstly, I convert the image to gray scale and then apply Gaussian blur to reduce the noise and to make the edges smooth as shown in Figure 9. Following that I used OpenCV canny edge detector to detect the edges of the parking space as shown in Figure 10. Then I used the probabilistic Hugh transform (PHT) to detect the lines from the image. Please note that I also tested with Standard Hugh Transform (SHT) which also works but it sometimes returns coordinates in negative format which makes image annotation complex for this task. Hence, I used PHT which also resulted in better line detection than SHT. I experimented with different threshold values here and 80 worked for my example image to detect all the lines as shown in Figure 11. The process up to this is demonstrated in Figure 09.

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
image_path = '/content/p4.png'
image = cv2.imread(image_path)
image = cv2.resize(image, (1000,400))
image1 = image.copy()
image2 = image.copy()

gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blurred_image = cv2.GaussianBlur(gray_image, (5, 5), 0) #smoothing the image to remove noise
edges = cv2.Canny(blurred_image, 70, 150, apertureSize=3)
linesP = cv2.HoughLinesP(edges, 1, np.pi / 180, threshold=80, minLineLength=30, maxLineGap=10)
```

Figure 09: loading parking space image and detecting lines

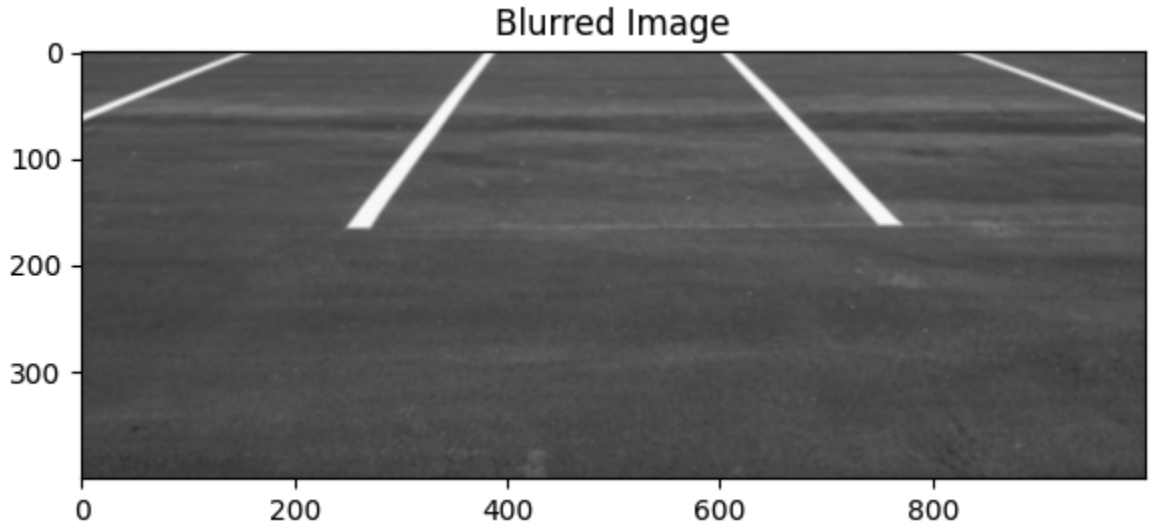

Figure 10: Converting image to grayscale and applying gaussian blur

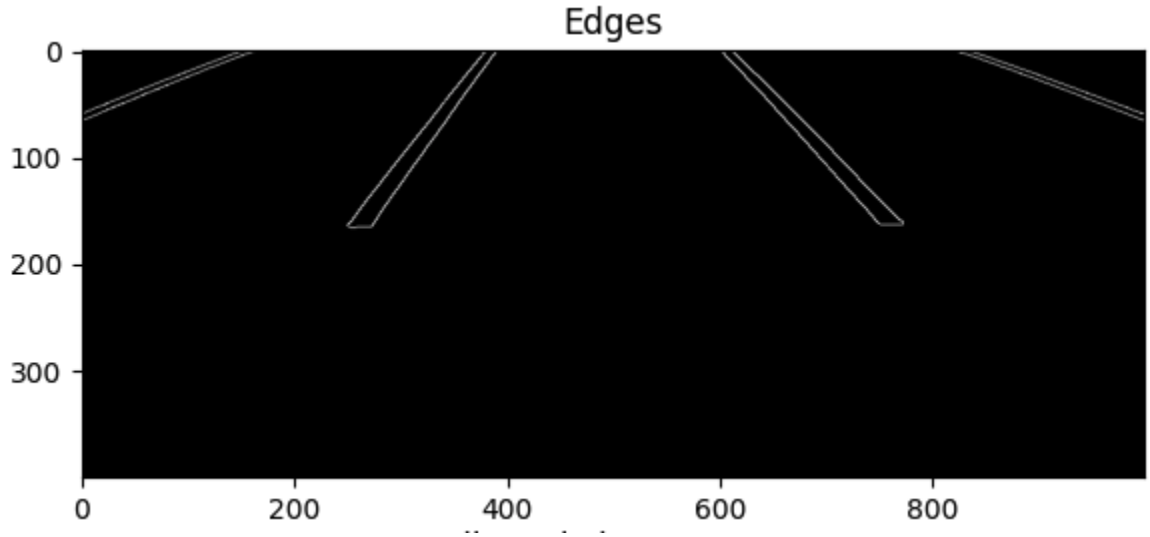

Figure 11: Detecting edges for the parking space Canny Edge

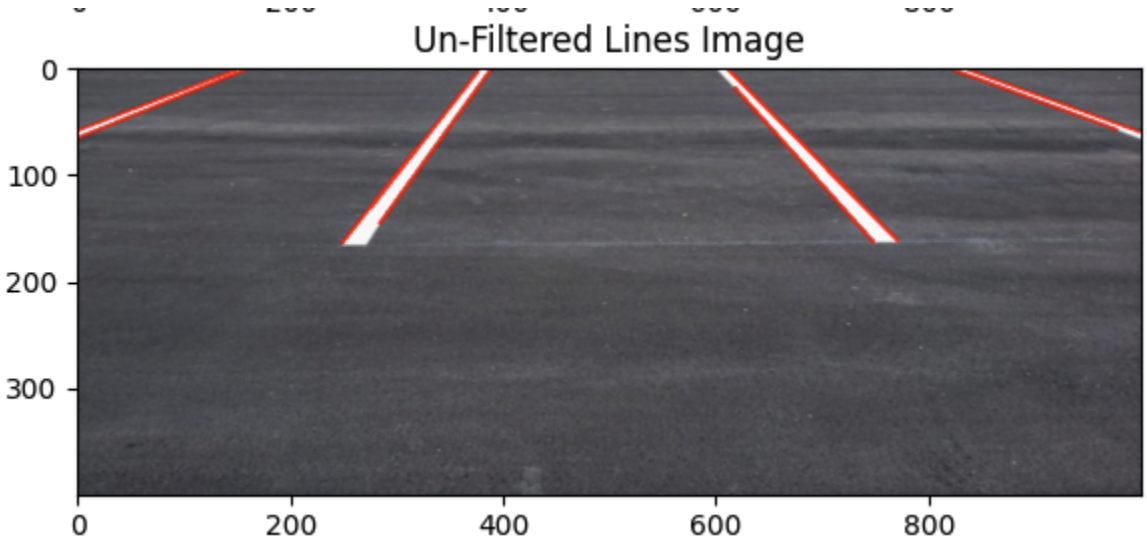

Figure 12: Detecting lines for the parking space using Hugh Transform

Extracting detected vertical line information and saving them for posterior analyzing like filtering lines to detect available parking space number, figure 13.

```
vertical_lines = []

if linesP is not None:
    for line in linesP:
        x1, y1, x2, y2 = line[0]

        # Calculation for the angle to filter vertical lines, if necessary
        angle = np.arctan2(y2 - y1, x2 - x1) * 180.0 / np.pi

        if -60 < abs(angle) < 100 or 300 < abs(angle) < 280:
            vertical_lines.append([x1, y1, x2, y2])
            cv2.line(image1, (x1, y1), (x2, y2), (0, 0, 255), 2)

print(f'Initial number of detected lines: {len(vertical_lines)}')
```

Figure 13: Extracting line information and drawing over the image

Once we have all the detected line information we can filter them by checking if the two lines are really close. If two lines are close we omit them and only consider a free parking space if two lines have a minimum distance of 70. In this way are only taking one line from each parking border and this helps us to filter out only one line for each border. I also do one additional check if multiple lines are really close but if the line is small the algorithm should ignore that and take the longest vertical line.

```
min_distance = 70

def line_length(line):
    return ((line[2] - line[0])**2 + (line[3] - line[1])**2)**0.5

vertical_lines_sorted = sorted(vertical_lines, key=lambda x: x[2])

filtered_lines = [vertical_lines_sorted[0]]

for i in range(1, len(vertical_lines_sorted)):
    current_line = vertical_lines_sorted[i]
    previous_line = filtered_lines[-1]

    distance = abs(current_line[2] - previous_line[2])
    if distance > min_distance:
        filtered_lines.append(current_line)
    else:
        # If the lines are too close, compare their lengths and keep the longer one
        if line_length(current_line) > line_length(previous_line):
            filtered_lines[-1] = current_line

print(f'Final number of detected lines after filtering: {len(filtered_lines)}')
print('TOTAL FREE PARKING SPACE: ', len(filtered_lines) - 1)
```

```
Initial number of detected lines: 10
Final number of detected lines after filtering: 4
TOTAL FREE PARKING SPACE:  3
```

Figure 13: Filtering the detected edges for the parking space

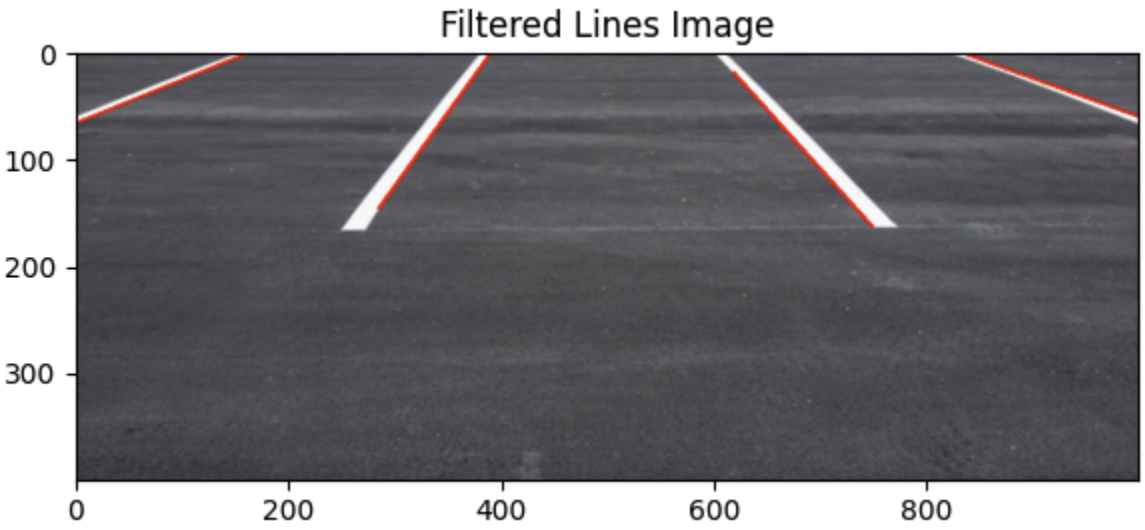

Figure 14: Filtering the detected edges for the parking space

Once I have the filtered lines I draw the lines over the image and also draw the parking space numbers sequentially on the image as shown in Figure 15-16.

```
54 for line in filtered_lines:
55   # print(line)
56   x1, y1, x2, y2 = line
57   cv2.line(image, (x1, y1), (x2, y2), (0, 0, 255), 2)
58
59 for idx, line in enumerate(filtered_lines[:-1], start=1):
60     next_line = filtered_lines[idx]
61     mid_x = (line[0] + next_line[0]) // 2
62     cv2.putText(image2, str(idx), (mid_x, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)
```

Figure 15: Drawing and Detecting the number of available parking space

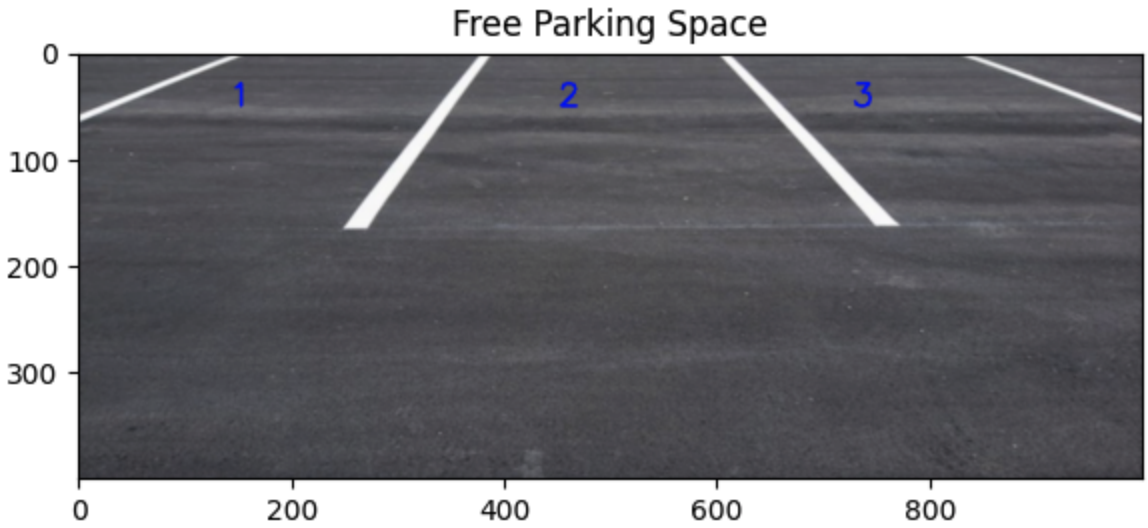

Figure 16: Detecting the number of available parking spaces

Figure 17 shows the overall output pictures of the whole algorithm.

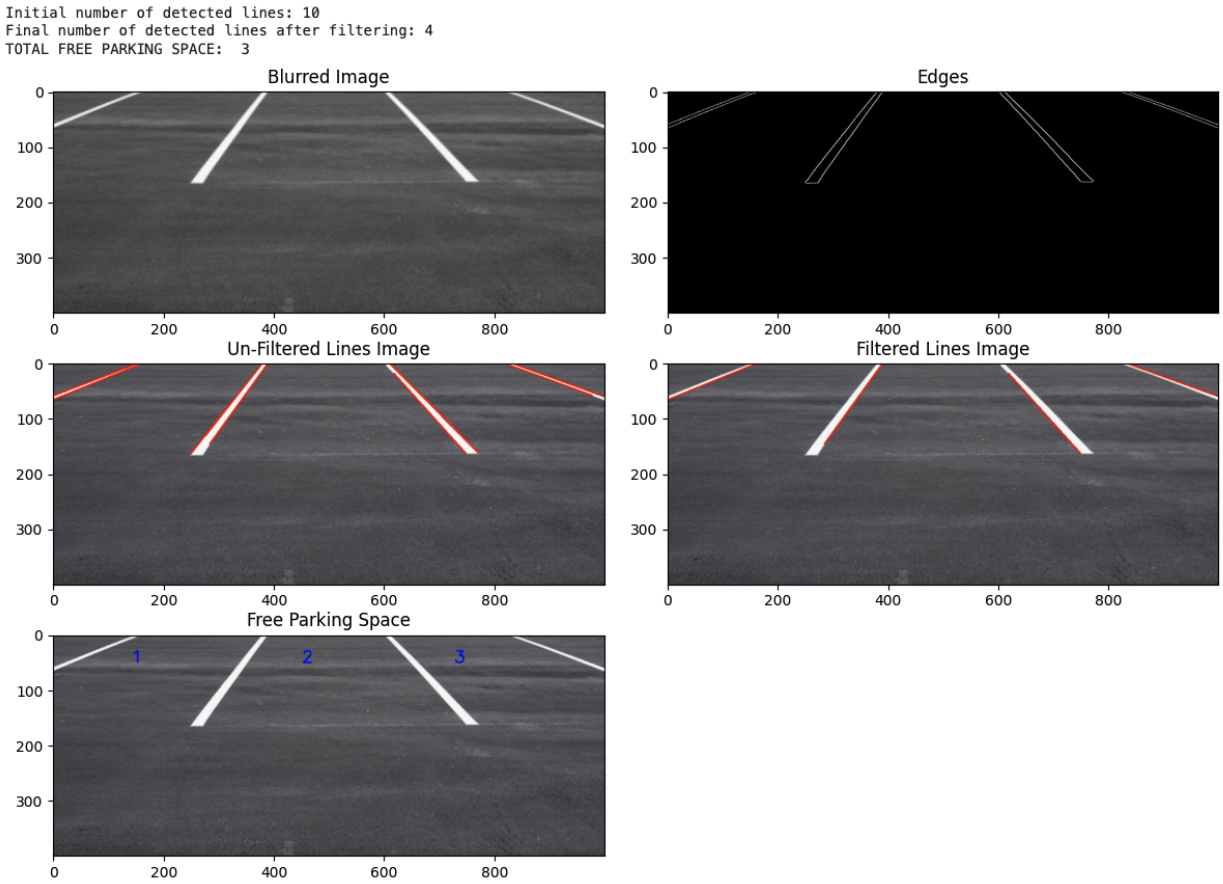

Figure 16: Output picture of the algorithm

Conclusions:

This report shows how to detect a face from any image, find a match with a random image with a face, and return a score of the matched images using the OpenCV library. Also, this report explores the usage of the OpenCV library in real-world examples of detecting free parking space from an image. Even though detecting lines from any given image was easy, this task was challenging as we had to filter out lines as per our requirement and detect free parking space. I hope to explore more on this and add more complex scenarios (i.e. detecting parking space from video, when there are cars, etc.) to this assignment on my own to build up my skills on OpenCV.

Appendix A: Source code

Task 01:

```
import cv2
from pathlib import Path
from random import choice

class FaceMatcher:
    def __init__(self, data_path):
        self.data_path = Path(data_path)
        self.faces_path = self.data_path / "faces"
        self.faces_path.mkdir(exist_ok=True)
                                self.face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')
        self.orb = cv2.ORB_create()
        self.bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

    def load_image(self, file_path):
        image = cv2.imread(str(file_path))
        if image is None:
            print(f"Image {file_path} not found or unable to load.")
            return None
        return image

    def detect_faces(self, image):
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        faces = self.face_cascade.detectMultiScale(gray, 1.1, 4)
        return [image[y:y+h, x:x+w] for (x, y, w, h) in faces]

    def save_faces(self, faces, image_number):
        face_paths = []
        for idx, face in enumerate(faces):
            face_filename = f"face{image_number}_{idx}.png"
            face_path = self.faces_path / face_filename
            cv2.imwrite(str(face_path), face)
            face_paths.append(face_path)
        return face_paths

    def find_keypoints_descriptors(self, image):
        gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

```
        keypoints, descriptors = self.orb.detectAndCompute(gray_image, None)
        return keypoints, descriptors

    def match_faces(self, descriptors_to_match, image_paths):
        best_match_path = None
        best_match_score = -1
        for path in image_paths:
            image = self.load_image(path)
            if image is not None:
                keypoints, descriptors = self.find_keypoints_descriptors(image)
                if descriptors is not None and len(descriptors) > 0:
                    matches = self.bf.match(descriptors_to_match, descriptors)
                    matches = sorted(matches, key=lambda x: x.distance)
                    match_score = len(matches)
                    if match_score > best_match_score:
                        best_match_score = match_score
                        best_match_path = path
        return best_match_path, best_match_score

# Defining root dir
data_path = "/root"
# Creating FaceMatcher object to initialize all the required methods
face_matcher = FaceMatcher(data_path)

# Detecting and saving faces from images
face_images_paths = []
for i in range(1, 11):
    image_path = face_matcher.data_path / f"image{i}.png"
    if not image_path.is_file():
        image_path = face_matcher.data_path / f"image{i}.jpg"
    image = face_matcher.load_image(image_path)
    if image is not None:
        faces = face_matcher.detect_faces(image)
        face_paths = face_matcher.save_faces(faces, i)
        face_images_paths.extend(face_paths)

# Selecting a random face and finding its descriptions
random_face_path = choice(face_images_paths)
random_face_image = face_matcher.load_image(random_face_path)
_, descriptors_to_match = face_matcher.find_keypoints_descriptors(random_face_image)
```

```
# Matching the face descriptors against all original images
original_image_paths = [str(face_matcher.data_path / f"image{i}.png") if (face_matcher.data_path / f"image{i}.png").is_file() else
                        str(face_matcher.data_path / f"image{i}.jpg") if (face_matcher.data_path / f"image{i}.jpg").is_file() else None
                    for i in range(1, 11)]
original_image_paths = [path for path in original_image_paths if path is not None]
best_match_path, best_match_score = face_matcher.match_faces(descriptors_to_match, original_image_paths)

print(random_face_path, best_match_path, best_match_score)
print("Match_score: ", best_match_score)
# print(best_match_path)
random_face_path = random_face_path
best_match_path = best_match_path
random_img = cv2.imread(str(random_face_path))
best_match = cv2.imread(str(best_match_path))

plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(random_img, cv2.COLOR_BGR2RGB))
plt.title('Randomly Selected Image')
plt.show()
plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(best_match, cv2.COLOR_BGR2RGB))
plt.title('Matched Facial Image')
plt.show()
```

Task 02:

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
image_path = '/content/p4.png'
image = cv2.imread(image_path)
image = cv2.resize(image, (1000,400))
image1 = image.copy()
image2 = image.copy()

gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

```
blurred_image = cv2.GaussianBlur(gray_image, (5, 5), 0) #smoothing the image to remove noise
edges = cv2.Canny(blurred_image, 70, 150, apertureSize=3)
linesP = cv2.HoughLinesP(edges, 1, np.pi / 180, threshold=80, minLineLength=30, maxLineGap=10)

vertical_lines = []

if linesP is not None:
    for line in linesP:
        x1, y1, x2, y2 = line[0]

        # Calculation for the angle to filter vertical lines, if necessary
        angle = np.arctan2(y2 - y1, x2 - x1) * 180.0 / np.pi

        if -60 < abs(angle) < 100 or 300 < abs(angle) < 280:
            vertical_lines.append([x1, y1, x2, y2])
            cv2.line(image1, (x1, y1), (x2, y2), (0, 0, 255), 2)

print(f'Initial number of detected lines: {len(vertical_lines)}')

min_distance = 70

def line_length(line):
    return ((line[2] - line[0])**2 + (line[3] - line[1])**2)**0.5

vertical_lines_sorted = sorted(vertical_lines, key=lambda x: x[2])

filtered_lines = [vertical_lines_sorted[0]]

for i in range(1, len(vertical_lines_sorted)):
    current_line = vertical_lines_sorted[i]
    previous_line = filtered_lines[-1]

    distance = abs(current_line[2] - previous_line[2])
    if distance > min_distance:
        filtered_lines.append(current_line)
    else:
        # If the lines are too close, compare their lengths and keep the longer one
        if line_length(current_line) > line_length(previous_line):
```

```
            filtered_lines[-1] = current_line

print(f'Final number of detected lines after filtering: {len(filtered_lines)}')
print('TOTAL FREE PARKING SPACE: ', len(filtered_lines) - 1)

for line in filtered_lines:
  # print(line)
  x1, y1, x2, y2 = line
  cv2.line(image, (x1, y1), (x2, y2), (0, 0, 255), 2)

for idx, line in enumerate(filtered_lines[:-1], start=1):
    next_line = filtered_lines[idx]
    mid_x = (line[0] + next_line[0]) // 2
    cv2.putText(image2, str(idx), (mid_x, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (255,
0, 0), 2)



numbered_image_dir = '/content/parking_space_numbered.png'
cv2.imwrite(numbered_image_dir, image)



plt.figure(figsize=(12, 8))
plt.subplot(3, 2, 4)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Filtered Lines Image')
plt.subplot(3, 2, 3)
plt.imshow(cv2.cvtColor(image1, cv2.COLOR_BGR2RGB))
plt.title('Un-Filtered Lines Image')

# Display blurred image in 2nd subplot (bottom left)
plt.subplot(3, 2, 1)
plt.imshow(cv2.cvtColor(blurred_image, cv2.COLOR_BGR2RGB))
plt.title('Blurred Image')

# Display edges in 3rd subplot (bottom right)
plt.subplot(3, 2, 2)
plt.imshow(cv2.cvtColor(edges, cv2.COLOR_BGR2RGB))
plt.title('Edges')
```

```
plt.subplot(3, 2, 5)
plt.imshow(cv2.cvtColor(image2, cv2.COLOR_BGR2RGB))
plt.title('Free Parking Space')

# plt.subplot(3, 2, 4)
# plt.imshow(cv2.cvtColor(lines, cv2.COLOR_BGR2RGB))
# plt.title('Edges')

plt.tight_layout()  # Adjust layout to make room for titles and ensure no overlap
plt.show()
```