Creating Activity Timeline Using Computer Vision

Sabin K. Pradhan

May 1, 2018

Abstract: Often during video surveillance, video footage is recorded and in event of a crime, investigators have to go through entire clips concerning the estimated time line of the video to find when the crime happened. More often than not, investigators waste their time looking over clips that have no human activity. This project looks to improve their research ability by using computer vision to identify high human activity intervals in surveillance footage.

Contents

1 Introduction

Working as an after-hour Administrator for Residence Life, I have had to handle multiple instances of theft. More often than not, the campus safety surveillance team has these incident recorded in video, but, usually takes some time for the officers to study multiple hours worth of surveillance footage to identify the perpetrator. This project looks to use the Open-source Computer Vision Library in Python to study surveillance footage, and construct a database that reflects an activity timeline. This information will then be imported into Tableau to create interactive visualization to easily identify when the human activity occurred.

2 Methodology

For this project, the video is recorded via the laptops web camera and uses *CascadeClassifier* class in *cv2* Python module to create an object to upload a frontal_face_haarcascade.xml file. The program then uses this object to detect faces in the video stream and time stamp when faces enter and exit the video frame. The project plans to inject this data into a *psychopg2* database which is connected to Tableau. Interactive visualizations are then built on Tableau to seamlessly identify human activity in the footage.

2.1 Preliminaries

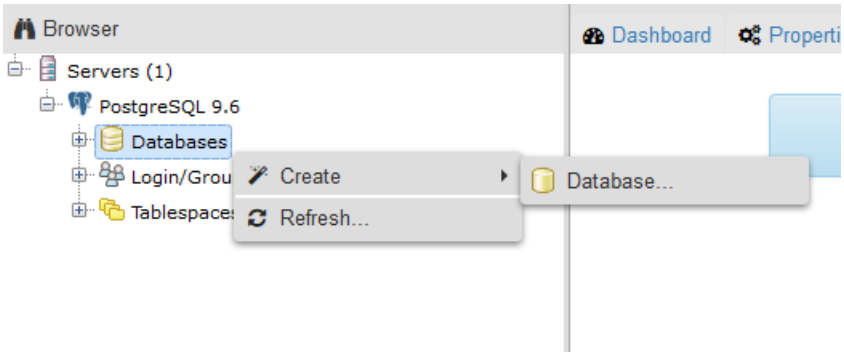
Before moving ahead to the code, be sure to download the frontal face haarcascade xml file which holds the facial feature data that helps the model detect faces.

In addition, download the Python's cv2 and the *psycopg2* module from the links before proceeding further.

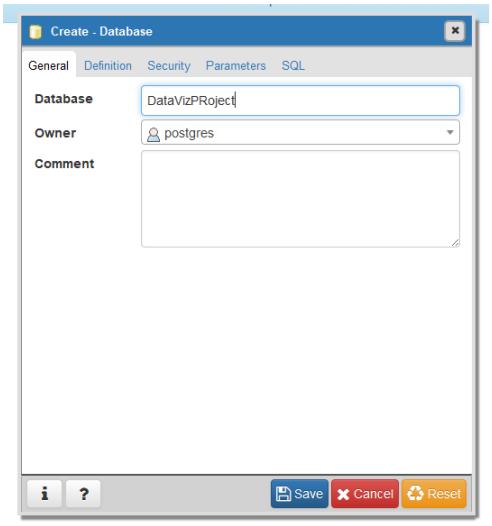
Note: A working version of PostgreSQL needs to be downloaded before proceeding further.

2.2 Creating Database in PostgreSQL

First create a database in PostgreSQL named *DataVizProject*. Right click on 'Databases', then click on 'Create'> 'Database'. This can be easily done using the PostgreSQL UI as shown below:



Fill in the details for the database *DataVizProject*, then click Save.



2.3 Importing Libraries

The program starts by importing the above mentioned libraries.

Python Program:

```
import cv2
from datetime import datetime
import psycopg2
import postgresid as pg
```

The *prostgresid* module is a self created module that holds the PostgreSQL server credential.

2.4 Setting Up Psycopg2 Module

The program first creates a PostgreSQL database connecter using the *connect* command. Using the credentials from the *postgresid* module, the program access into the database and connects via port 5432 which is the default PostgreSQL access point. The program then uses the *cursor* object *c* to allow us to execute SQL commands through python.

Python Program:

```
#creating connector
conn=psycopg2.connect(
        "dbname='DataVizProject' user='"+
        pg.username+
        "' password='"+
        pg.password+
        "' host='localhost' port='5432'")
c=conn.cursor()
```

Once the program connects to the database and creates the cursor object, it creates the table *face_ activity* (if it does not already exist) to store the data generated by the program, namely the number of faces in the frame and the time stamp.

Python Program:

```
# Create table
c.execute("CREATE TABLE if not exists face_activity (date
↪  timestamp , number_of_faces int)")
```

2.5 Setting Up OpenCV2 Module

To set up the *cv2* module, first the program creates a *CascadeCassifier* object *face_ cascade* which takes in the *haarcascade_ frontalface_ default.xml* file to create a face detection model.

Python Program:

```
face_cascade=cv2.CascadeClassifier("haarcascade_frontalface_default.xml")
```

Additionally, the program also sets up a *VideoCapture* object *vid* to take data from the web camera using the by passing the parameter 0.

Python Program:

```
vid=cv2.VideoCapture(0)
```

2.6 Main Program

The program defines the variables *prev_face_num* and *frame_num* before creating a continuous *while* loop to keep tabs of the changes in the number of faces displayed in the frame and the number of frames the program processes should one need to jump frames. The user may choose to exit the loop by pressing 'q'.

Python Program:

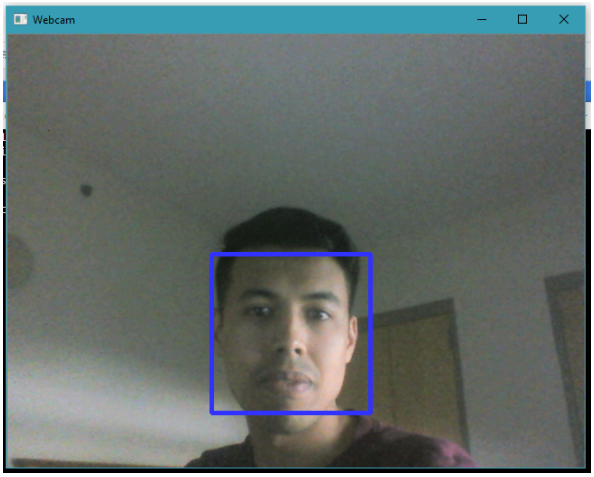
```
prev_face_num=None
frame_num=0
print("Press q to quit.")
```

The *while* starts with an update to the *frame_num*. Then the program calls the *read* method from the *vid* object which returns a frame captured by the web camera stored in the variable *frame* along with a boolean that checks if the camera is still online stored in *check*.

A copy of the frame is then converted to a gray scale image and stored in *img_gray* using the *cvtColor* function. The program convert the image to gray scale as it works better with the haarcascade files to detect faces.

The program then uses the *detectMultiScale* method from the *CascadeClassifier* class and uses a sliding window to inspect *img_gray* to see if it can detect any faces. This method returns a list containing tuples with the x coordinate, y coordinate, width and height of where the method detects the face. By storing the length of this list in the variable *cur_face_num* the program gets the number of faces the algorithm detects on the screen. Whenever the number of faces in the scree changes, the program executes an insert statement that adds a row to the PostgreSQL database table.

Then, using the *rectangle* function the program adds rectangles on the original *frame* around the faces detected by the algorithm. The algorithm can be made more sensitive by tuning the *scaleFactor* in the *detectMultiScale* method (E.g. 1.02).



The *imshow* function opens an iPython window and displays the *frame* originally captured by the camera along with the newly drawn rectangles. The *waitKey* function allows the user to close the iPython window. More importantly, when *waitKey* is called with the parameter 1, it allows the user to store an input *key*, which the program later uses to see if the *while* loop should be ended.

Python Program:

```
while True:
        frame_num+=1
        check, frame = vid.read()

        img_gray=cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)

        faces=face_cascade.detectMultiScale(img_gray,
         ↪  scaleFactor=1.05, minNeighbors=5)

        cur_face_num=len(faces)

        if cur_face_num!=prev_face_num and cur_face_num!=0:
            #inserts new row into table
                c.execute("INSERT INTO face_activity VALUES
                 ↪  ('"+str(datetime.now())+"',"+str(cur_face_num)+")")

        for x,y,w,h in faces:
                frame= cv2.rectangle(frame, (x,y),
                 ↪  (x+w,y+h),(255,50,50),3)

        cv2.imshow("Webcam",frame)
        key=cv2.waitKey(1)

        prev_face_num=cur_face_num

        if key==ord('q'):
                break
```

2.7 Closing Module

Once the program has finished running, it closes it's connections to the web camera using the *release* method. On the other hand, the iPython web camera window is shutdown using the *cv2.destroyAllWindows()* command. In addition, the program saves all the additional rows using the *commit()* method before severing connections with the PostgreSQL database using the *close()* method.

Python Program:

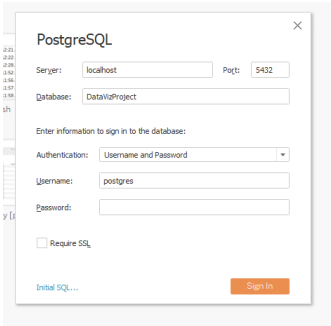
```
#closing connection to camera
```

```
vid.release()
#closes all ipython windows
cv2.destroyAllWindows()
# Save (commit) the changes
conn.commit()
# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be
↪  lost.
conn.close()
```

3 Tableau Visualization Results

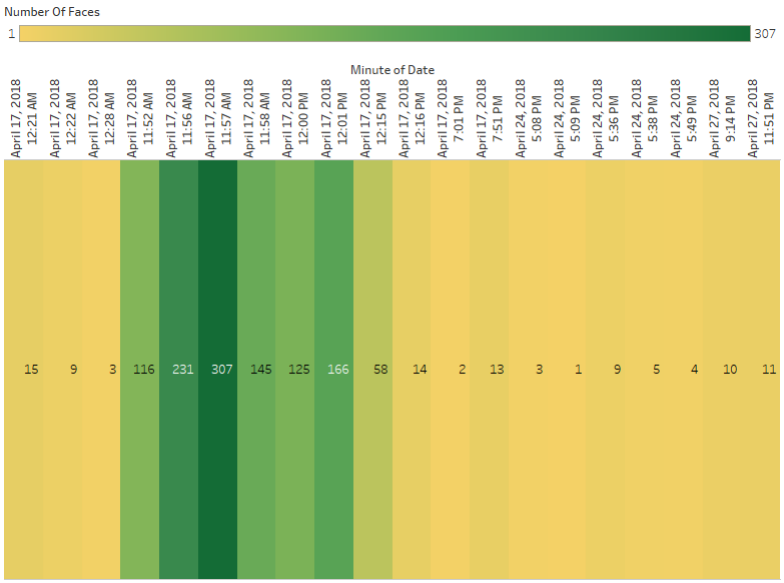
3.1 Setting Up

Now that all of the data from the program has been committed to a PostgresQL database table, the project uses Tableau's PostgreSQL connector to easily import and create visualizations. To do this, once Tableau is open, click on PostgreSQL under Connect to a Server. The following window should appear.



3.2 Visualization Dashboard

The data pulled from the PostgreSQL is then plotted in a time series heat map and presented in the dashboard as shown below. As the legend shows, the frames that had a high number of faces are shown in a darker green color while yellow denotes less activity in the video.



4 Conclusion

Using this methodology, the program was able to differentiate between frames with low activity versus frames with high activity. In addition, the program discarded frames without any facial activity.

4.1 Limitation and Improvements

One of the current limitation of this project is that to detect human activity, it needs to detect their faces. This is not always feasible, especially in cases where the perpetrators intentionally cover their faces.

In such cases, if more haarcascade files were added to the program to recognize other body features, this could help improve the robustness of the program. Having the ability to detect multiple human body parts would allow the model to function efficiently in cases where the faces were covered.

In addition, we could use facial recognition to the faces we detect to identity people with a past history of crimes and be aware of their presence. However, this poses an ethical dilemma relating to personal privacy that is beyond the purview of this paper.