Lie Detection Using Machine Vision Technologies and Techniques

The Islanders

Avlonitis Ektoras – 216882

Gaitanis Pablos – 265879

Lazarević Miloš – 272036

Vavakas Alexandros – 272483

Abstract

This project is a lie detector created using Machine Vision technologies and methods. Haar Cascades, being the backbone of this project, are used in every implementation of the lie detector, to avoid preprocessing steps that could be detrimental in later computations. The participants in the testing process included the developer team consisting of four members, in addition to one opportunity of a live demonstration. This project involved multiple failed attempts, successes, and discoveries that contributed to the development of the lie detector. In addition, background research and exploration of different approaches were pivotal in deciding the direction of the project.

Listing the methods used, the approach taken to achieve the goal is outlined, including, but not limited to, assumptions, drawbacks, and the justification of each step. Furthermore, additional approaches both inside and outside the Machine Vision scope are discussed, showing the limitations of this project and what could be done in the future to improve it, should the resources arise to support that direction. Finally, a conclusion is reached, based on the work done in a limited amount of time, allowing for an estimation whether further contributions are necessary to make the concept of lie detection using machine vision techniques feasible.

Table of Contents

List of Figures

Introduction

Overview

Lie detection is not a new concept in the industry. The practice of lie detection via the use of scientific tools was introduced by Cesare Lombroso, an Italian criminologist that applied a blood pressure instrument (hydrosphygmograph) to criminal suspects in 1895. Later, in 1921, John Augustus Larson developed a machine that could record physiological responses to questioning during interrogation of criminal suspects.[1] These instruments were called "Polygraphs" which were incorrectly referred to as the "lie detector test".[2] The topic of lie detection has been explored in various ways, from some of the newest neurotechnology such as recording electromagnetic signals from the brain[3] to the more old-fashioned psychological interrogation[4].

Further implementations of lie detection system using tools involves machine learning. Machine learning models are trained to capture behavioral cues such as eye movement, gestures, speech, and other physical indicators. A machine learning lie detector could be made to work universally by taking a sample of the participant population and having them train the model which is then used to detect lies in the rest of the participant population. It is noteworthy that psychological factors can decrease the accuracy of this detection, which is why focusing the interpretation on the eyes and speech may be the more favorable option.[5]

Taking inspiration from existing research, we will be using technologies and methods from the Machine Vision scope exclusively, to analyze facial features from static images, video frames and live video. The target: attempting to identify if a person is lying by discerning the differences in facial features such as the eyes, mouth, and the face of that person in previous encounters.

The Machine Vision scope does not include features which may be crucial to the lie detection concept, however, for the purposes of this project we employ all Machine Vision techniques and methods that can apply in this scenario.

The following pages will discuss each method used and its purpose in this project. The key technologies and techniques will be outlined, explaining their role in the lie detector and how they combine to make the final product work, concluding on the limitations of this project.

[1] Palmiotto, M J. "Historical Review of Lie-Detection Methods Used in Detecting Criminal Acts." Historical Review of Lie-Detection Methods Used in Detecting Criminal Acts | Office of Justice Programs, https://www.ojp.gov/ncjrs/virtual-library/abstracts/historical-review-lie-detection-methods-used-detecting-criminal#:~:text=The%20first%20attempt%20to%20use,(hydrosphygmograph)%20to%20criminal%20suspects.
[2] "Polygraph." Wikipedia, Wikimedia Foundation, 2 Dec. 2022, https://en.wikipedia.org/wiki/Polygraph.
[3] Wolpe, Paul Root, Kenneth R. Foster, and Daniel D. Langleben. "Emerging neurotechnologies for lie-detection: Promises and perils." *The American Journal of Bioethics* 5.2 (2005): 39-49.
[4] Trovillo, Paul V. "History of lie detection." *Am. Inst. Crim. L. & Criminology* 29 (1938): 848.
[5] Gonzalez-Billandon, Jonas, et al. "Can a Robot Catch You Lying? A Machine Learning System to Detect Lies during Interactions." Frontiers, Frontiers, 12 July 2019, https://www.frontiersin.org/articles/10.3389/frobt.2019.00064/full.

Haar Cascades

Haar Cascades are object detection algorithms that can be used to identify eyes, mouth, face, and other features of a human. One of the ways they do this is by using edge detection and line detection. More importantly, the algorithms are given a plethora of positive images, where in terms of face detection, a face is shown, and an equal number of negative images, where the face is not shown. It is trained to detect faces using these samples.[6]

Additionally, like a kernel or matrix, Haar Cascades use Machine Vision features to scan through an image to find these facial components:

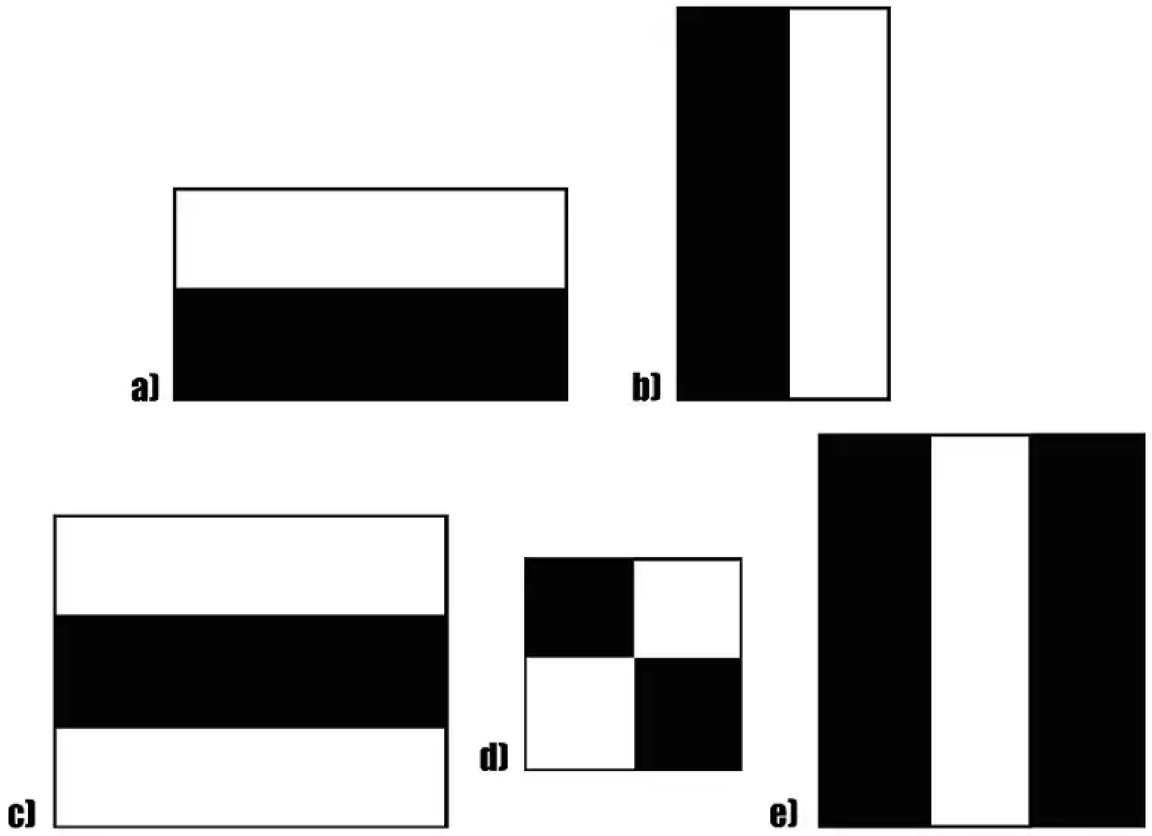


Figure 1. DIFFERENT TYPES OF HAAR CASCADE FEATURES

Haar Cascades are the backbone of our project, upon which all other techniques we used are applied.

[6] Behera, Girija Shankar. "Face Detection with Haar Cascade." Medium, Towards Data Science, 29 Dec. 2020, https://towardsdatascience.com/face-detection-with-haar-cascade-727f68dafd08.

The ABC Method

The ABC method is a recurring theme in our project. It concerns how our lie detector concludes that an individual is lying. The concept is straightforward: Using a sample of a person's face when telling the truth (**A**) and a sample when telling a lie (**B**) the lie detector would determine the unknown scenario (**C**). The requirement is that the lie detector needs to be calibrated with A and B before it can determine C. In addition, A and B need to be sampled in a controlled environment including, but not limited to, lighting, angle, and distance. Otherwise, the noise in the video/image can be falsely flagged as part of the facial features.

Theory of Lie Detection

A lie detector is a system that is used to detect deception or dishonesty. It typically works by measuring physiological responses, such as changes in heart rate, blood pressure, and respiration, to determine whether a person is being truthful or deceptive.

There are a few different approaches that can be used to create a lie detector using Python. One approach is to use machine learning algorithms to analyze the video stream and look for subtle changes in the person's facial expressions, body language, or voice that might indicate deception. For example, the system could use Python libraries such as OpenCV or scikit-learn to analyze the video stream and identify features such as changes in the person's eye movements, facial muscles, or tone of voice that are commonly associated with lying.

There are several factors that can affect the accuracy of a lie detector, including the specific techniques and algorithms used, the training and experience of the person administering the test, and the individual characteristics of the person being tested.

Another approach is to use sensors to measure the person's physiological responses, such as changes in heart rate or blood pressure, and use these measurements to determine whether the person is being truthful or deceptive. This approach can be combined with the use of machine learning algorithms to analyze the video stream to provide a more accurate prediction of whether the person is lying. Python libraries such as NumPy or SciPy can be used to analyze the sensor data and identify patterns or trends that may indicate deception.

Materials and Methods

First Step: Video Frames

Recording the Videos

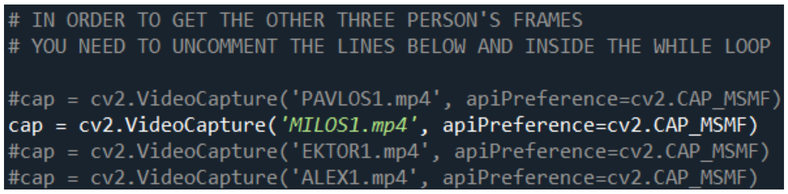
At first, we took four examples, one for each team member, to try this project test. For each one we recorded a video, where we asked the team member to tell one truth and one lie. For the first seconds of the video, the person was authentic and his face more relaxed, as he was telling the truth. For the last seconds of the video, there is apparent change in his facial expression, because of the lie. To minimize the errors, each member cooperates, and truthfully tells the truth and the lie. Also, for the scope of this project, we assume that when the person was lying the facial expressions change is clear.

Therefore, we ended up with four six-second mp4 file videos, one for each member. These mp4 files were then imported to the python code created for collecting frames from the videos.

The videos that we obtained had to be retaken many times because bad lighting conditions, wrong distance from the camera, and looking away, caused many problems in the second step of the project, when using the Haar cascades. The problems that we dealt with are going to be discussed below in each section.

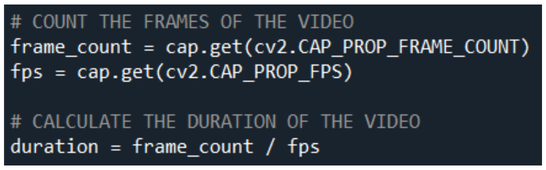
Collecting Frames

The only library needed for this part was *cv2*. Firstly, we imported the video and then, counted the frames and its duration.



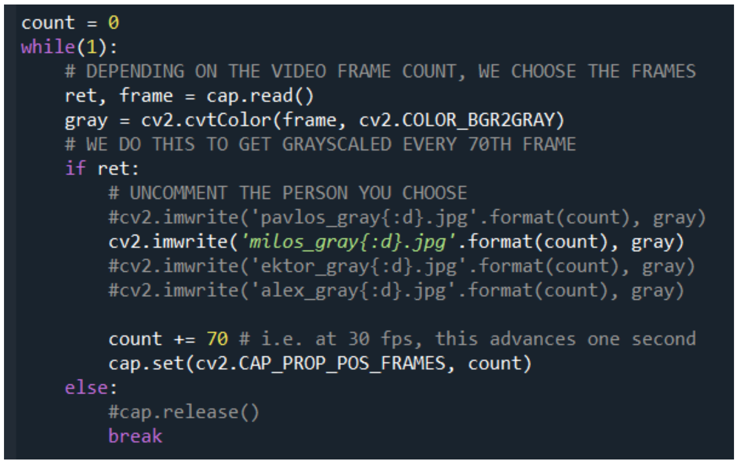
```
# IN ORDER TO GET THE OTHER THREE PERSON'S FRAMES
# YOU NEED TO UNCOMMENT THE LINES BELOW AND INSIDE THE WHILE LOOP

#cap = cv2.VideoCapture('PAVLOS1.mp4', apiPreference=cv2.CAP_MSMF)
cap = cv2.VideoCapture('MILOS1.mp4', apiPreference=cv2.CAP_MSMF)
#cap = cv2.VideoCapture('EKTOR1.mp4', apiPreference=cv2.CAP_MSMF)
#cap = cv2.VideoCapture('ALEX1.mp4', apiPreference=cv2.CAP_MSMF)
```

As we observed, each video duration was about 6 – 7 seconds, whereas the frame count depended on the video and the camera. For the three videos the frames where 180 – 200, whereas one video only captured 100 frames. To be more precise, we should have same video cameras and same environment for the frame count to be similar and thus for the python code to work automatically on everyone.



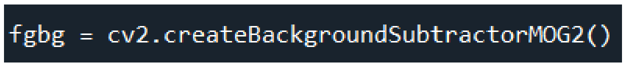
```
# COUNT THE FRAMES OF THE VIDEO
frame_count = cap.get(cv2.CAP_PROP_FRAME_COUNT)
fps = cap.get(cv2.CAP_PROP_FPS)

# CALCULATE THE DURATION OF THE VIDEO
duration = frame_count / fps
```

After that, we moved on to the *while* loop where we read each frame, turned it to grayscale, and then captured and saved the frame using the *cv2.imwrite* function. We captured every 70th frame including the first (0th), to collect a frame from the first three seconds of the video and the last three seconds of the video. For the one video with less frames, the 0th and 70th frames where enough to continue to our next python code with the cascades, and thus we didn't need to change anything.

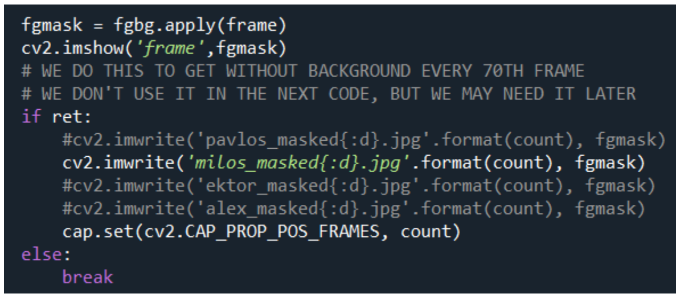


```python
count = 0
while(1):
    # DEPENDING ON THE VIDEO FRAME COUNT, WE CHOOSE THE FRAMES
    ret, frame = cap.read()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # WE DO THIS TO GET GRAYSCALED EVERY 70TH FRAME
    if ret:
        # UNCOMMENT THE PERSON YOU CHOOSE
        #cv2.imwrite('pavlos_gray{:d}.jpg'.format(count), gray)
        cv2.imwrite('milos_gray{:d}.jpg'.format(count), gray)
        #cv2.imwrite('ektor_gray{:d}.jpg'.format(count), gray)
        #cv2.imwrite('alex_gray{:d}.jpg'.format(count), gray)

        count += 70 # i.e. at 30 fps, this advances one second
        cap.set(cv2.CAP_PROP_POS_FRAMES, count)
    else:
        #cap.release()
        break
```

For future reference and just for curiosity we thought of another way to collect frames that may eliminate the need of some work in the next code. We thought of background subtraction, to compare the two different facial expressions, using the *cv2.createBackgroundSubtractorMOG2()* and then saving frames the same way as we did before. However, this was not the way we proceeded but we took a glance of how this could have been done that way.



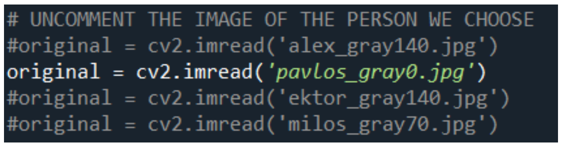
```python
fgbg = cv2.createBackgroundSubtractorMOG2()
```



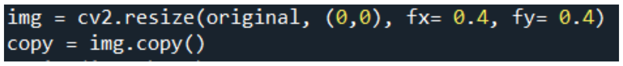
```python
fgmask = fgbg.apply(frame)
cv2.imshow('frame',fgmask)
# WE DO THIS TO GET WITHOUT BACKGROUND EVERY 70TH FRAME
# WE DON'T USE IT IN THE NEXT CODE, BUT WE MAY NEED IT LATER
if ret:
    #cv2.imwrite('pavlos_masked{:d}.jpg'.format(count), fgmask)
    cv2.imwrite('milos_masked{:d}.jpg'.format(count), fgmask)
    #cv2.imwrite('ektor_masked{:d}.jpg'.format(count), fgmask)
    #cv2.imwrite('alex_masked{:d}.jpg'.format(count), fgmask)
    cap.set(cv2.CAP_PROP_POS_FRAMES, count)
else:
    break
```

Second Step: Cascades

Firstly, we imported the frames in the new python script using the *cv2.imread* function. Whatever the frame's size was, we used the *cv2.resize* to resize it to our chosen coordinates.

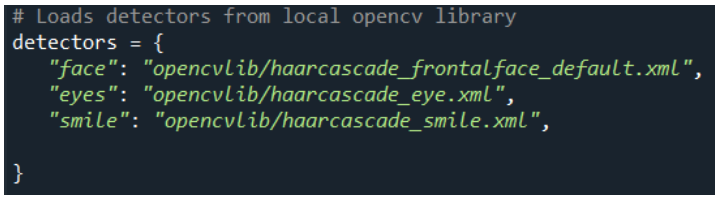


```
# UNCOMMENT THE IMAGE OF THE PERSON WE CHOOSE
#original = cv2.imread('alex_gray140.jpg')
original = cv2.imread('pavlos_gray0.jpg')
#original = cv2.imread('ektor_gray140.jpg')
#original = cv2.imread('milos_gray70.jpg')
```



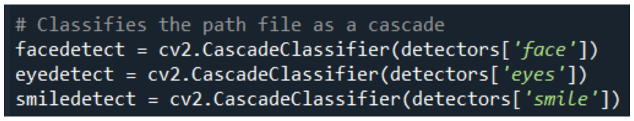
```
img = cv2.resize(original, (0,0), fx= 0.4, fy= 0.4)
copy = img.copy()
```

We loaded the detectors ("face", "eyes", "smile") from a local OpenCV library.



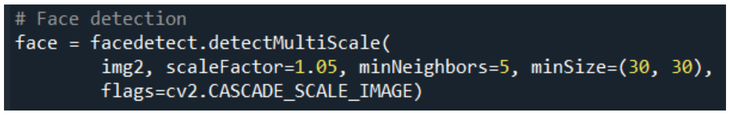
```
# Loads detectors from local opencv library
detectors = {
   "face": "opencvlib/haarcascade_frontalface_default.xml",
   "eyes": "opencvlib/haarcascade_eye.xml",
   "smile": "opencvlib/haarcascade_smile.xml",

}
```

OpenCV offers some pretrained models that we read using the *cv2.CascadeClassifier* for the detections of the face, eyes, and smile.



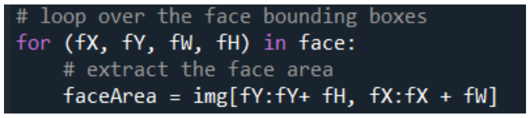
```
# Classifies the path file as a cascade
facedetect = cv2.CascadeClassifier(detectors['face'])
eyedetect = cv2.CascadeClassifier(detectors['eyes'])
smiledetect = cv2.CascadeClassifier(detectors['smile'])
```

We use those plus the *detectMultiScale* to perform the detections. The first thing we detected was the face.

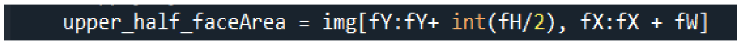


```
# Face detection
face = facedetect.detectMultiScale(
        img2, scaleFactor=1.05, minNeighbors=5, minSize=(30, 30),
        flags=cv2.CASCADE_SCALE_IMAGE)
```

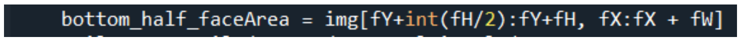
Afterwards we detected the eyes and the smile. We iterated through the face bounding box, so that there were no false eye and smile detections outside the face area. In order to increase the reliability of the detection, we created two new bounding boxes for the eyes and the smile respectively. We chose the upper half face area for the eye detection, and the bottom half face area for the smile detection, thus making it truthful and trustworthy.



```
# loop over the face bounding boxes
for (fX, fY, fW, fH) in face:
    # extract the face area
    faceArea = img[fY:fY+ fH, fX:fX + fW]
```

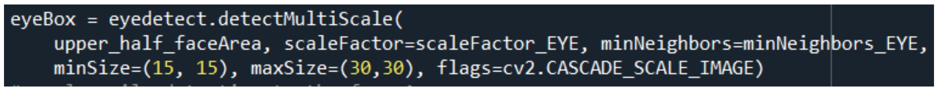


```
    upper_half_faceArea = img[fY:fY+ int(fH/2), fX:fX + fW]
```

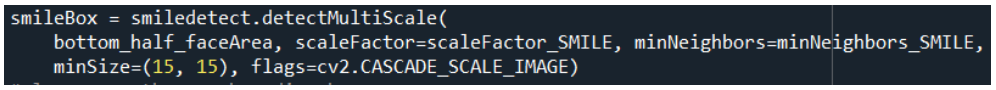


```
    bottom_half_faceArea = img[fY+int(fH/2):fY+fH, fX:fX + fW]
```

Using the cascades that from above, we searched for the eyes and the smile using the *eyedetect.detectMultiScale* and *smiledetect.detectMultiScale* functions. We had many parameters which need adjustments, and we figured out that for each different face they vary.



```
eyeBox = eyedetect.detectMultiScale(
    upper_half_faceArea, scaleFactor=scaleFactor_EYE, minNeighbors=minNeighbors_EYE,
    minSize=(15, 15), maxSize=(30,30), flags=cv2.CASCADE_SCALE_IMAGE)
```



```
smileBox = smiledetect.detectMultiScale(
    bottom_half_faceArea, scaleFactor=scaleFactor_SMILE, minNeighbors=minNeighbors_SMILE,
    minSize=(15, 15), flags=cv2.CASCADE_SCALE_IMAGE)
```

The *minNeighbors* parameter shows how many neighbors each candidate rectangle should have to retain it. Therefore, it made big difference. For example, in Figure 2 Alex's smile detection using *minNeighbors* =10 versus using *minNeighbors* = 15 resulted in the following:

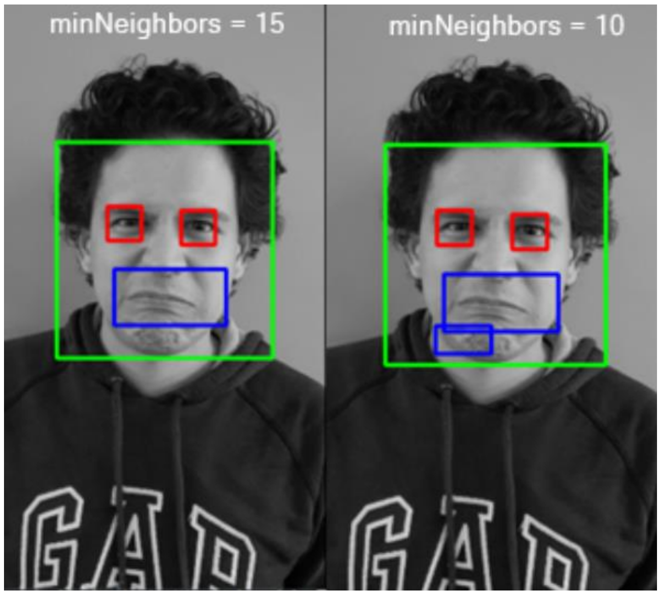


Figure 2 ALEX'S SMILE DETECTION FOR TWO DIFFERENT VALUES OF minNEIGHBORS

Another parameter we had to define for each picture was the *scaleFactor*, which specifies how much the image size is reduced at each image scale.

Another one was adding the *maxSize* parameter in the *eyeBox*. For example, in Figure 3 Milos eyes detections with and without the *maxSize* parameter are shown below:

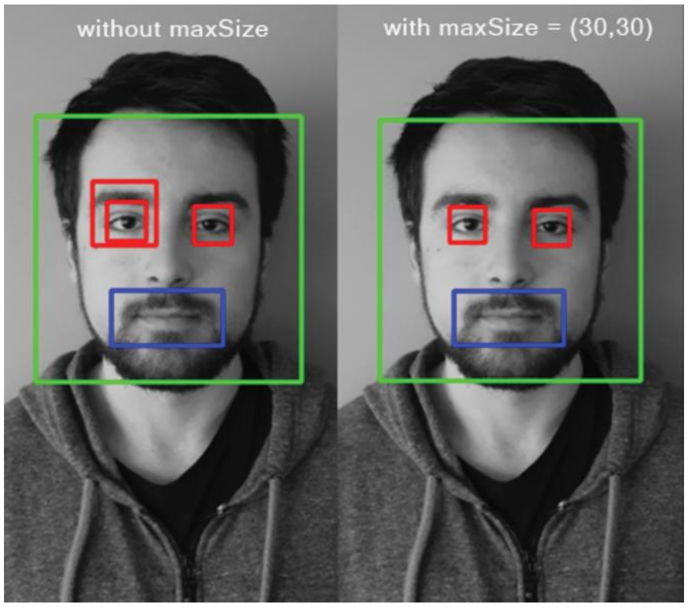
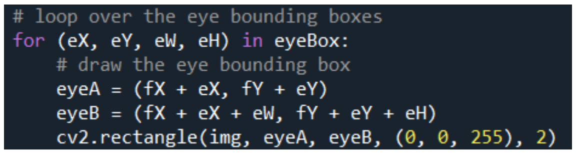
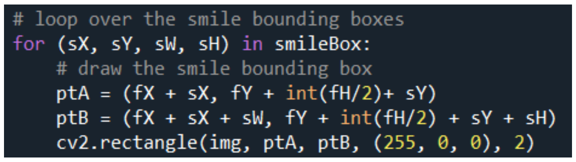


Figure 3 MILOS EYE DETECTION WITH AND WITHOUT USING THE MAXSIZE PARAMETER

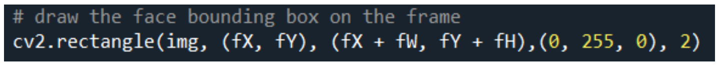
After those specifications we iterated through the *eyeBox* and *smileBox* that we created to draw the rectangles we want and display them (red color for the eyes, blue for the smile, green for the face).



```
# loop over the eye bounding boxes
for (eX, eY, eW, eH) in eyeBox:
    # draw the eye bounding box
    eyeA = (fX + eX, fY + eY)
    eyeB = (fX + eX + eW, fY + eY + eH)
    cv2.rectangle(img, eyeA, eyeB, (0, 0, 255), 2)
```



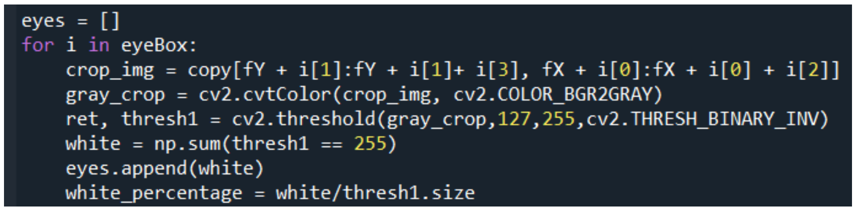
```
# loop over the smile bounding boxes
for (sX, sY, sW, sH) in smileBox:
    # draw the smile bounding box
    ptA = (fX + sX, fY + int(fH/2)+ sY)
    ptB = (fX + sX + sW, fY + int(fH/2) + sY + sH)
    cv2.rectangle(img, ptA, ptB, (255, 0, 0), 2)
```



```
# draw the face bounding box on the frame
cv2.rectangle(img, (fX, fY), (fX + fW, fY + fH),(0, 255, 0), 2)
```

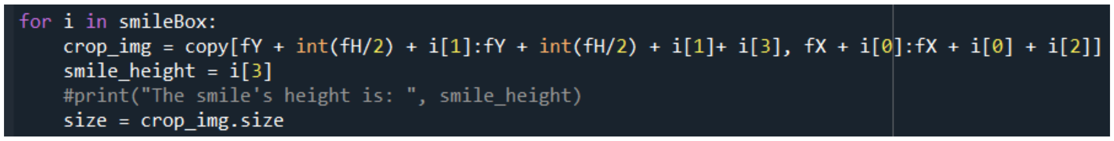
We had many failed attempts when trying to detect the desired features. That's why through trial and error we thought of these solutions to try and minimize for each candidate the wrong outputs.

Next, we move to the results. We want to understand for each individual how the facial expressions change, that is, how the two frames differ, when the person is telling the truth and when he is lying. To do that, we calculate the changes in two regions: the eyes and the smile. For the eyes, we calculated the white pixels inside the eye box. To do that, we made a copied image of the eye box, converted it to grayscale and then used the inverse binary threshold on it. We calculated the white pixels inside that box, and the percentage the white pixels are of that box's size. The fewer the white pixels, the less the eye is open, because as we observed from the frames, the area around the eyeball is converted to black. Another idea we had was that, instead of doing an inverse binary threshold for the eye, we could calculate another way more precisely the white pixels of the eyeball, meaning that we will just be calculating accurately just the white regions inside of the eye.



```python
eyes = []
for i in eyeBox:
    crop_img = copy[fY + i[1]:fY + i[1]+ i[3], fX + i[0]:fX + i[0] + i[2]]
    gray_crop = cv2.cvtColor(crop_img, cv2.COLOR_BGR2GRAY)
    ret, thresh1 = cv2.threshold(gray_crop,127,255,cv2.THRESH_BINARY_INV)
    white = np.sum(thresh1 == 255)
    eyes.append(white)
    white_percentage = white/thresh1.size
```

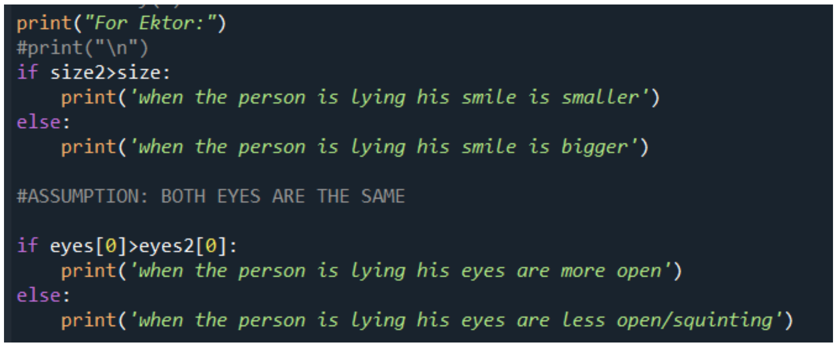
For the smile, we calculated the size of the smile box. The smaller the size, the less the person is smiling. In our case, we assume that the smaller the size of the smile box, the less the person is making any expressive mouth movement.



```python
for i in smileBox:
    crop_img = copy[fY + int(fH/2) + i[1]:fY + int(fH/2) + i[1]+ i[3], fX + i[0]:fX + i[0] + i[2]]
    smile_height = i[3]
    #print("The smile's height is: ", smile_height)
    size = crop_img.size
```

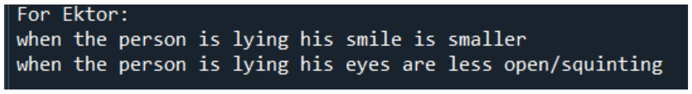
We used the copied images to show the eyes and the smile when we ran the code, in order to help us improve it.

Later, as we followed the same process for both the frames, we added the white pixels of each eye in an *eye* and *eye2* list for each frame, and the smile sizes were saved in two variables *size* and *size2*. We compared the first eyes for each frame, and the smile sizes as seen below:



```python
print("For Ektor:")
#print("\n")
if size2>size:
    print('when the person is lying his smile is smaller')
else:
    print('when the person is lying his smile is bigger')

#ASSUMPTION: BOTH EYES ARE THE SAME

if eyes[0]>eyes2[0]:
    print('when the person is lying his eyes are more open')
else:
    print('when the person is lying his eyes are less open/squinting')
```

That way we display for each candidate the results in the kernel. For example, for Ektor we found out that:



```
For Ektor:
when the person is lying his smile is smaller
when the person is lying his eyes are less open/squinting
```

We concluded that there is still a long way to go in order to improve our lie detector. Apart from the eyes and the smile, we can examine multiple other face features that may change from one frame to another and indicate a truth or lie. For example, the nose, the eyebrows, or even sudden head movements.

Third Step: Lie Detector in Live Video

“A Lie Detector should be able to identify if someone is lying at real time”

Our next step was to create a live lie detector, combining the methods we have already learned and used.

In our project we use OpenCV library to capture video frames from a webcam or other video stream, and then uses a machine learning model to make a prediction. The model that we use is pre-trained on a dataset of labeled video frames and is used to analyze each new frame as it is captured, making a prediction, and displaying the result on the screen in real-time. It uses Haar cascades to detect the eyes, face, and smile in the frame. The Haar cascades are pre-trained machine learning models that are designed to identify specific features in images, such as eyes, faces, and smiles. The code draws rectangles around the detected features and displays the frame with the rectangles on the screen. The code turns the frames of the live video in GrayScale for better identification of the face, eyes, and smile. It checks if the eyes are closed, and a smile is detected. If both conditions are met, the code displays the message "LIE" on the frame.

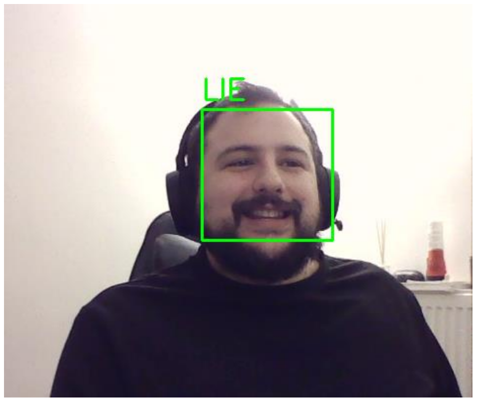


Figure 4 LIVE VIDEO DETECTION

In Figure 4, notice how the eyes are detected as closed due to the angle and height of the eyes.

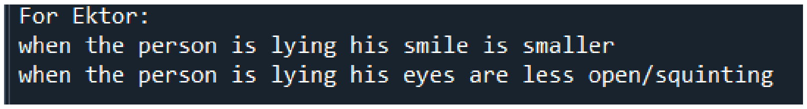
The general steps of our code are:

- The code captures a frame from the video stream using the cv2.VideoCapture() function.
- The frame is preprocessed by converting it to grayscale using cv2.cvtColor() functions.
- The code then uses the cv2.rectangle() function to draw rectangles around the detected features. This function takes as input the frame, the top-left corner and bottom-right corner of the rectangle, and the color and thickness of the rectangle.
- The code checks if the eyes are closed and a smile is detected. If both of these conditions are met, the code displays the message "LIE" on the frame.
- The code then displays the frame on the screen using the cv2.imshow() function. This function takes as input the name of the window and the frame to be displayed.
- The code checks if the user pressed the 'q' key to exit the loop. If the 'q' key was pressed, the loop breaks and the video capture device is released, and all windows are closed.

Results and Discussion

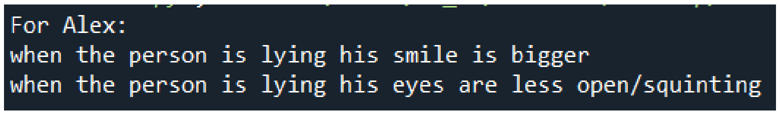
As mentioned above, for each candidate, we want to understand how the facial expressions change, that is, how the two frames differ, when the person is telling the truth and when he is lying. We display the results for each one:

For Ektor we found out that:



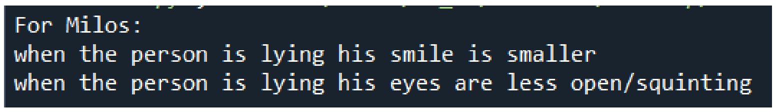
```
For Ektor:
when the person is lying his smile is smaller
when the person is lying his eyes are less open/squinting
```

For Alex we found out that:



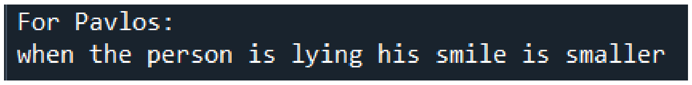
```
For Alex:
when the person is lying his smile is bigger
when the person is lying his eyes are less open/squinting
```

For Milos we found out that:



```
For Milos:
when the person is lying his smile is smaller
when the person is lying his eyes are less open/squinting
```

For Pablos we found out that:



```
For Pavlos:
when the person is lying his smile is smaller
```

As far as the live video lie detector is concerned, we have had the opportunity to test it with several participants during our demonstration.

Drawbacks

For the detections to work correctly, we had to record videos in a way that the face was clearly shown, symmetrical and without one side being brighter or darker than the other.

When we calculated the white pixels, we noticed some drawbacks due to the lack of keeping constant lighting conditions. When the light was off, there were some unnecessary shadows created in the area of the face due to the nose or other face features. This made the one eye darker than the other, and when we used the inverse binary threshold, the output was only white pixels, which was a problem. Thus, we had to record the videos again.

Also, when we recorded videos from a bigger distance, we had to change the parameters of the cascade detections. When the face was too far, the *maxSize* parameter had no value since the face was smaller and thus it captured every mistake again. Hence, the solution is to keep a constant distance from the camera or figure out another way to fix the cascade problem automatically for each person.

Another problem we encountered was when the person was looking away. As it can be seen from Pablo's example at Figure 5, the detector could not find his eyes. That could also occur, because of his eyes which were squinting when he smiled.

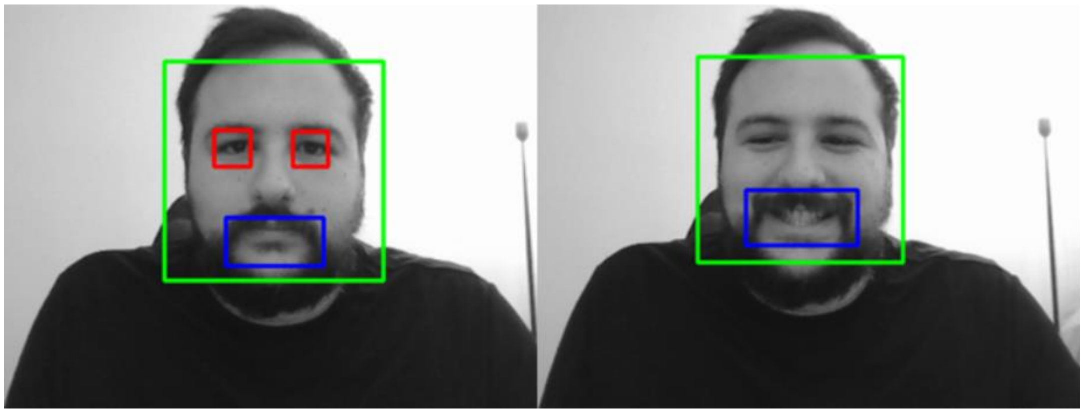


Figure 5 PABLO'S TWO FRAMES WITH ALL THE HAAR CASCADE DETECTIONS

The Haar cascades are not always able to accurately detect the eyes, smile, and face in every frame. This can lead to false positives, incorrectly detecting the features or false negatives failing to detect the features. Having the same light in the recording while capturing the frames could make a better example of detection. Haar cascades can be sensitive to changes in lighting and background conditions, which can affect their accuracy. This means that the lie detector may not perform consistently in different environments.

Furthermore, Haar cascades are limited to detecting specific features (eyes, mouths, smiles, faces) that have been trained into the model. This means that the lie detector may not be able to detect other cues that could indicate whether someone is lying, such as changes in body language or speech patterns.

Another limitation is that they require manual feature selection of the features to be detected, which can be time-consuming and requires expert knowledge.

Assumptions

When recording the videos, we assume that each member cooperated, and truthfully told the truth and the lie, to minimize the errors. Also, for the scope of this project, we assume that when the person was lying the facial expressions change was clear. When we displayed the results and compare the smile sizes, we assume that the smaller the size of the smile box, the less the person is making any expressive mouth movement.

For the program to understand which facial expressions links to which result it must be given the matching information from the subject.

Furthermore, since the program focuses on facial features, these characteristics should not deviate in form from the standard type of themselves. To elaborate, facial features such as missing teeth or eyes cannot be ignored and will be added to the program's calculation. Unfortunately, it currently is unable to perform on such occasions and they would alter the results. Thus, for the program to acquire only the right elements of the face, the face itself must be of "standard kind".

Finally, while the facial features remain unchanged in total in each video, their image must remain unchanged. Which leads to the requirement of the lighting conditions being similar if not the same in both the videos for calibration and the videos to be tested, because a change would alter the color of any given aspect exposed to such change.

Future Work

One of our main concerns is the differences in the quality of the video recordings. To be more precise, we should have for every recording, the same video cameras, and same environment for the frame count to be similar and thus for the python code to work automatically on everyone.

Another idea we had when we were using the cascades was that, instead of doing an inverse binary threshold for the eye, we could calculate another way more precisely the white pixels of the eyeball, meaning that we will just be calculating accurately just the white regions inside of the eye.

We concluded that there is still a long way to go in order to improve our lie detector. Apart from the eyes and the smile, we can examine multiple other face features that may change from one frame to another and indicate a truth or lie. For example, the nose, the eyebrows, or even sudden head movements.

For the live video component, there are some other preprocessing steps we could add to improve the accuracy of the Haar cascades in detecting the eyes, smile and face. For example:

- Enhancing the contrast of the frame: You can use image processing techniques such as histogram equalization or gamma correction to enhance the contrast of the frame, which can help the Haar cascades to detect the features more accurately.
- Applying a smoothing filter: You can use a smoothing filter such as a Gaussian blur to reduce the noise in the frame, which can also help the Haar cascades to detect the features more accurately.
- Cropping the frame: You can crop the frame to focus on a specific area where the features are likely to be located, which can help the Haar cascades to detect the features more accurately.

Some suggestions that could improve the performance of our models could be some other machine learning models, such as Deep learning models: Deep learning models, such as convolutional neural networks (CNNs), can be trained to detect various features in images, including eyes, mouths, smiles, and faces. These models are generally more accurate than Haar cascades but require more computational resources and a larger dataset to train on.

Another suggestion is the Machine learning algorithms, such as support vector machines (SVMs) or random forests, can be used to classify images based on the presence or absence of specific features. Same as the deep learning models, these algorithms may require more computational resources and a larger dataset to train on but can be more accurate than Haar cascades.

Our last suggestion would be Keras model, which is a high-level API for building and training deep learning models in Python, for building a lie detector. To do this, find below same steps very similar to our project:

- Collect and label a dataset of images that includes examples of people lying and not lying, along with the relevant features (eyes, mouths, smiles, faces).
- Preprocess the images in the dataset to prepare them for training the model. This may include steps such as resizing the images, converting them to grayscale, and normalizing the pixel values.
- Split the dataset into training, validation, and test sets.
- Define the architecture of the Keras model, which could be a convolutional neural network (CNN) or some other type of deep learning model.
- Compile the model with an appropriate loss function, optimizer, and metrics.

- Train the model on the training set, using the validation set to monitor the performance of the model and tune the hyperparameters.
- Evaluate the model on the test set to measure its performance.
- Use the trained model to classify new images as lying or not lying, based on the presence or absence of the relevant features.

To keep the main idea of our course, Machine Vision, we didn't use our own trained model, as mentioned above. We created this code, but instead of using a trained model to predict whether the person is lying, it uses the Haar cascade classifier to detect the person's smile and eyes in the video frame. If a smile is detected, but no eyes are detected, the code assumes that the person is lying. More preprocessing steps could be added on the video frame, so the classifier could detect more accurate the eyes and smile. A lie detection system could use multiple cues, such as facial expressions, body language, vocal cues, and contextual information, to make a more accurate prediction. But it could never be 100% accurate. As proven that a person could surpass a lie detector.

References

1) Palmiotto, M J. "Historical Review of Lie-Detection Methods Used in Detecting Criminal Acts." Historical Review of Lie-Detection Methods Used in Detecting Criminal Acts | Office of Justice Programs, https://www.ojp.gov/ncjrs/virtual-library/abstracts/historical-review-lie-detection-methods-used-detecting-criminal#:~:text=The%20first%20attempt%20to%20use,(hydrosphygmograph)%20to%20criminal%20suspects.
2) "Polygraph." Wikipedia, Wikimedia Foundation, 2 Dec. 2022, https://en.wikipedia.org/wiki/Polygraph.
3) Wolpe, Paul Root, Kenneth R. Foster, and Daniel D. Langleben. "Emerging neurotechnologies for lie-detection: Promises and perils." The American Journal of Bioethics 5.2 (2005): 39-49.
4) Trovillo, Paul V. "History of lie detection." Am. Inst. Crim. L. & Criminology 29 (1938): 848.
5) Gonzalez-Billandon, Jonas, et al. "Can a Robot Catch You Lying? A Machine Learning System to Detect Lies during Interactions." Frontiers, Frontiers, 12 July 2019, https://www.frontiersin.org/articles/10.3389/frobt.2019.00064/full.
6) Behera, Girija Shankar. "Face Detection with Haar Cascade." Medium, Towards Data Science, 29 Dec. 2020, https://towardsdatascience.com/face-detection-with-haar-cascade-727f68dafd08