Real-Time Face Mask Detection & People Counting System

John Georgousis

MSc Data Science
The University of Bath
2022

Real-Time Face Mask Detection & People Counting System

Submitted by: John Georgousis

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Master of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

In this dissertation we compare technologies for building a real-time face mask detection and people counting system and produce a working prototype that can be readily adopted by businesses. We compare the performances of 3 state-of-the-art face detectors and 2 mask classifiers which provides research and implementation insights. The best-performing models are chosen for the production of a complete application that is ready for use. The application is able to display live video information relating to face mask use and people count, and perform reliable data collection with an estimated accuracy of 100% in a simple use-case and 80% in more complex scenarios.

Contents

Acknowledgements

I would like to thank my supervisor, Dr Hongping Cai, for being open to my modifying her original project idea and for her super quick responses to any questions I had. In addition, I would like to thank my family for their excitement to participate in the testing process of my software.

Chapter 1

Introduction

1.1 Problem Description

The purpose of this project is to compare methodologies for building a real-time face mask detection and people counting system and to produce a prototype that is optimised for real-world applications. The final version of the system is able to process raw, live video data of people entering a space and output information relating to *people count* and *face-mask use*. An example output is shown in Figure 1.1.

1.2 Background and History

The two main utilities that the system has to offer, people counting and face mask detection, are independent ideas that have been heavily researched and implemented in the past[1] (Ding, Li and Yastremsky, 2021)

1.2.1 Face Mask Detection

Context

Face mask detection refers to the process of observing and analysing an image/video and identifying how many masked and non-masked faces appear in it. While computer vision applications have been relevant ever since the emergence of the field, face mask detection has exploded in popularity with the outbreak of the Coronavirus Disease in 2019, as shown in Figure 1.2 (Vibhuti et al., 2022). The World Health Organisation (WHO) highlights the importance of wearing a mask in controlling the contagious disease (Organization, 2021) while multiple countries have, at certain times, imposed strict face mask mandates (Review, 2022). This has resulted in the need for detecting and documenting face mask use in individuals, as required by government regulations.

Problem

The problem that arises is that manually tracking whether or not people are wearing face masks can be cumbersome. In most cases, a paid employee will have to observe the area of

[1] https://www.sciencegate.app/keyword/281293

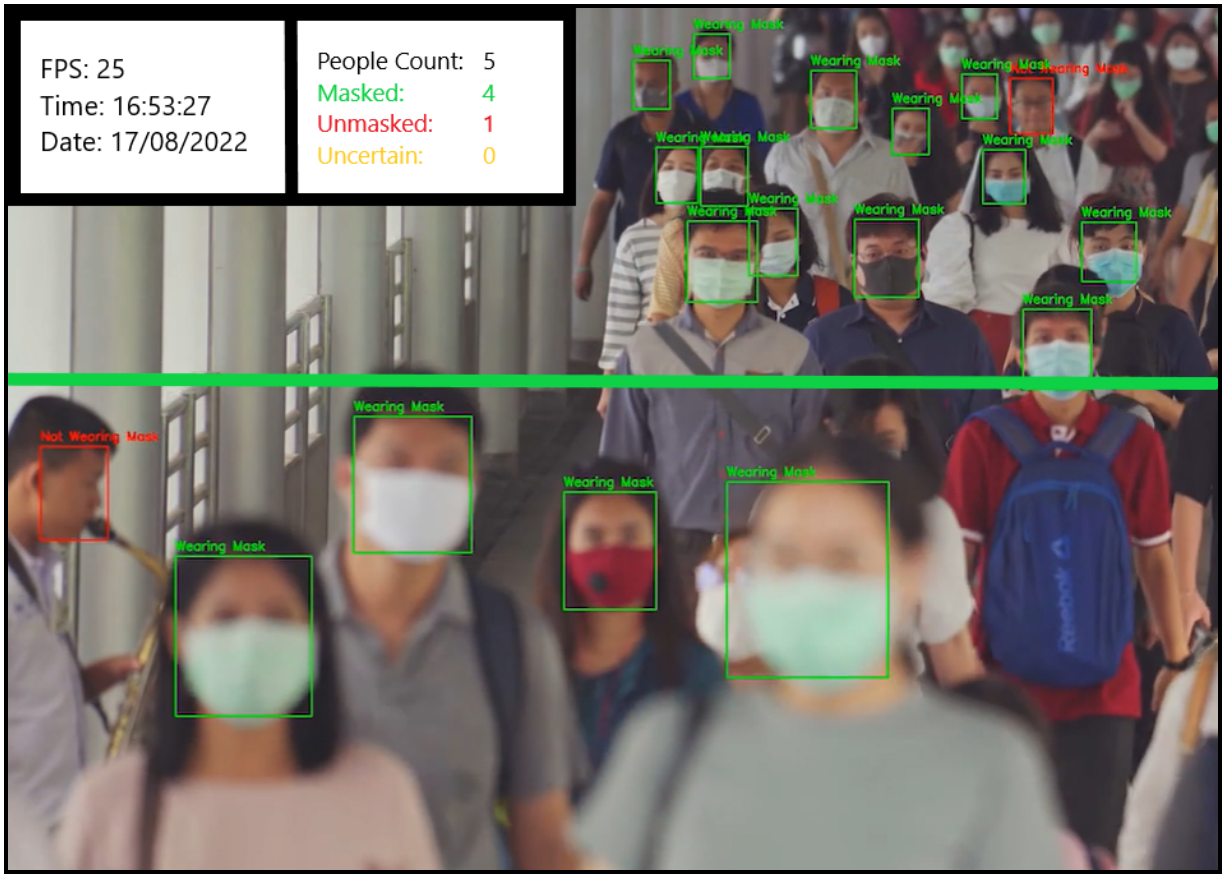

Figure 1.1: Example of our system's output stream. The horizontal, green line (boundary line) acts as a counter each time a face crosses it. The people count along with the face mask predictions are displayed in the top left corner along with other relevant information. (Edited image of Karchaudhuri (2020)).

interest (e.g. the entrance of a store) and consistently and reliably document every individual that enters the area during its operating hours. An additional step would also be needed to provide reports on the collected data. Such a process would be costly and inefficient. Thus, everyone from small retail businesses to large organisations could benefit from automation.

Solution

Our system offers the opportunity to automatically detect individuals of interest (e.g. customers), identify whether or not they are using a face mask, catalogue the results, and produce insightful reports on face-mask use. This utility has been combined with another in-demand feature: people counting.

1.2.2 People Counting

Context

People counting (also known as footfall counting, visitor counting, customer counting) refers to the use of a person or a system for the counting of individuals entering or leaving an area (e.g. a store, vehicle, public space, etc.). People counters have numerous benefits (LTD, 2022) such as:

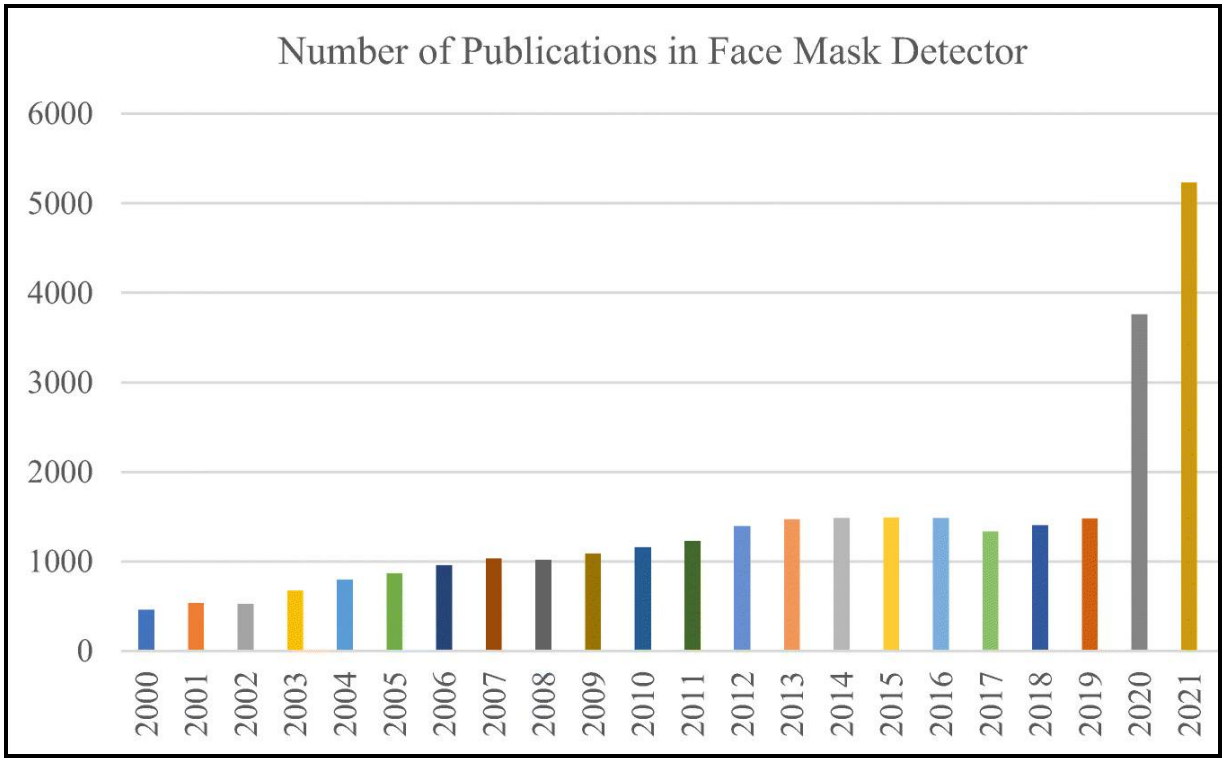

Figure 1.2: (Vibhuti et al., 2022)

1. Quantification of the success of marketing strategies by observing which one results in more customers.
2. Improvement of customer service by quickly responding to a spike in store visitors.
3. Statistical insights that allow for the analysis of long-term trends.
4. Increased safety and abiding by social distancing measures.
5. Comparison with other retail sites and competition.

Counting Technologies

Counting technologies range from manual clickers and beam counters that were common in the 1990s and the 2000s to more advanced methods such as camera-based and AI-Powered counters that have been increasing in popularity since 2018 (Metro, 2022) with the development of computer vision and deep learning techniques. Our system offers reliable, AI-Powered people counting using state-of-the-art deep learning models and tracking algorithms.

1.3 System Description

The system is a software application (executable file) that connects to a web camera and runs Python code. It consists of the following components:

1. **Face detector model**: Detects faces in the video input.
2. **Mask classification model**: Classifies faces as masked, unmasked, or uncertain.

3. **Object tracking and people counting library**: Tracks faces and catalogues detections.

The complete system pipeline is shown in Figure 1.3.

Figure 1.3: Diagram of our system's pipeline. The video stream outputs individual frames to be passed through the face detector. Then, the face detector outputs images of faces which are the input of the mask classifier. The classifier's predictions along with the locations of the faces are input in our tracking and counting library that outputs information and statistics relating to face mask use and people count.

1.4 Related Work

While a considerable amount of research has been conducted on face mask detection and people counting individually, we were not able to find an implementation that combined the two within the same system. Thus, in this section, we will discuss separate face mask detection and people counting implementations.

1.4.1 Face Mask Detection

A wide array of face mask detection systems very similar to our own have been implemented and are already being sold to businesses. Unfortunately, a request for a demo of these software applications was declined by the creators, so this section will reference only what the systems claim to do (without explicit references to accuracy and other metrics).

In LeewayHertz (2022), LeewayHertz developed face mask detection software that detects masked/unmasked individuals, generates reports, real-time statistics, and alerts (much like our system). One favourable addition is the implementation of a dashboard that serves as a high-level graphical user interface for monitoring purposes. Very similar implementations were produced across most companies that we examined (Meraki, 2022; Eocortex, 2022; Minolta, 2022), as well as Sightcorp (by Raydiant, 2022) which reported a "98% accuracy rate" on a reduced MAFA dataset. All implementations offer the same features: a built-in user interface, and notification/report generation.

1.4.2 People Counting

People counting is a much older idea than face mask detection and involves more diverse technologies including light beams, sound analysis, video analysis, and more. In this section, the technologies closest to our implementation will be mentioned that involve camera-based counting. Academic implementations are numerous[2], but here we will focus on more relevant

[2]https://paperswithcode.com/task/crowd-counting

people counting products that have been developed.
In two companies, (Camlytics, 2022; Cloudmatrix, 2022; Techvision, 2021), software was implemented that connects to an IP camera, counts people, and outputs statistics and visualisations on a built-in user interface. Both these implementations rely on AI-based object detection and tracking, much like our own, with the exception that body detection is employed rather than face detection. Certain unique features encountered involve the addition of automated emails from the system, connecting to a phone application, and gender/age estimation of customers. More people counting implementations can be found here: Slashdot (2022).

1.5 Objectives and Deliverables

This project is mainly a software development project in its implementing and testing a complete system that can be used by companies in the real world. In addition to this, it is a research project since it aims to explore more ways to improve face mask detection and people counting by enabling the comparison of architectures and models via controlled experiments. In this section, we will describe the main objectives of the project.

System Completeness

The most important objective is to implement a complete system that can easily be adopted and used by any business. The system should be capable of both real-time face mask detection and people counting, as well as the collection of data at a minimum.

Experiments & System Optimisation

A crucial objective of any real-time face mask detection system and people counting system is its performance. Key performance metrics include its accuracy and speed. This project will explore which methods are most effective at optimising inference speed and accuracy, providing important insights into this field of research as well as reliable data collection and minimum hardware requirements for the user.

Features for Real-World Applications

Another important objective of this project is to make this system beneficial in the real world. This will be partly determined by its prediction time and accuracy, but also due to added features that are rarely present in academic implementations. Such features include an intuitive user interface with live statistics, data collection & periodic export for analysis, and data visualisation.

Evaluation

The project will be evaluated on the insights it provides into improving system performance in this field, as well as its suitability for and benefits in real-world applications. In academic machine learning, one of the most commonly used metrics to describe performance is a model's accuracy on the test set. However, our system has much more complex goals since it is a combination of many components, so it will be tested on real-world situations of people entering an area under natural conditions. The system's performance on these tests best reflects its capabilities and success in any intended use-case.

Chapter 2

Background Theory and Technology Survey

In this chapter, we will cover theoretical background and implementation methodologies that have been utilised in this field along with their strengths and weaknesses. We will also identify ways to extend these methodologies and describe how our new developments can be beneficial. The topics will include:

1. Object Detection
2. Mask classification
3. Object Tracking
4. People Counting

Note: Most of these topics have extensive background theory and thus will be described more briefly to ensure that all important information will be covered within the given word limit.

2.1 Object Detection: One-Stage vs Two-Stage

In the first step of our system's pipeline, video frames are passed through an object detector to localise all faces in the frame. In this section, we will explore the two main classes of deep learning object detectors: one-stage and two-stage detectors.

2.1.1 Haar Cascade Methods

As a historic note, it should be mentioned that along with deep learning object detection, there are older object detection methods such as the Haar Cascade methods. These methods use traditional machine learning rather than state-of-the-art neural networks to learn a cascade function that can be accurate and efficient in face detection[1]. However, this project will focus on deep learning object detectors since they have always been more accurate, and certain variations of them can be faster as well (Andrie Asmara, Ridwan and Budiprasetyo, 2021).

[1] https://docs.opencv.org/3.4/db/d28/tutorial$_c$ascade$_c$lassifier.html

2.1.2 Deep Learning Object Detection: One-Stage vs Two-Stage

In deep learning object detection, there are two main types of object detectors: two-stage detectors and one-stage detectors. In two-stage detectors, as the name suggests, object detection is performed in two stages, and there is usually a higher accuracy rate (Soviany and Ionescu, 2018). On the other hand, one-stage detectors look at the image only once for detection, and they are typically faster than two-stage detectors (Soviany and Ionescu, 2018). Popular, state-of-the-art two-stage detectors include Faster R-CNN and Mask R-CNN (Soviany and Ionescu, 2018), but when it comes to live video object detection, inference speed can be more favourable than accuracy. Current state-of-the-art one-stage detectors such as YOLO (Redmon and Farhadi, 2018) and SSD (Liu et al., 2016) have been shown to be highly accurate two-stage detectors in video face mask detection applications (Ding, Li and Yastremsky, 2021; Yadav, 2020; Braulio Rios, 2020; Goyal, 2022). Coupling this with their significantly faster speeds to those of two-stage detectors makes them an intuitive pick for this project's implementation.
Next, we will explore the two main ways to combine object detection and image classification which will allow us to complete a face mask detection system.
Note: Two separate topics.
There are two main tasks of our system: 1. Face mask detection and 2. People counting.
It should be noted that these are two independent tasks (each with their own body of research) and thus will be discussed separately.

2.2 System Architectures

In this section, we will discuss system architectures relating to the first main task of our system (face mask detection), which allow us to combine detection and classification. We will also make this specific to our use-case involving face masks in video streams.

2.2.1 Two Main Architectures

In a nutshell, object detection systems work by processing input data (e.g. images) and producing outputs relating to object location and classification. There are two generic system architectures (Ding, Li and Yastremsky, 2021) that accomplish face mask detection (shown in Figure 2.1):

1. Using a face detector that takes video frames as input and outputs arrays of faces which are then input to a separate mask image classifier for classification.
2. Using an object detection network to perform localisation and classification in one step.

In related work (Ding, Li and Yastremsky, 2021; Yadav, 2020; Braulio Rios, 2020; Goyal, 2022), there have been comparisons of the two approaches showing that both can be highly effective for real-time face mask detection in video data.

2.2.2 Architecture #1

On the first approach of a single object detection network, the researchers concluded that it is likely to outperform the second approach due to its superior inference speed and accuracy.

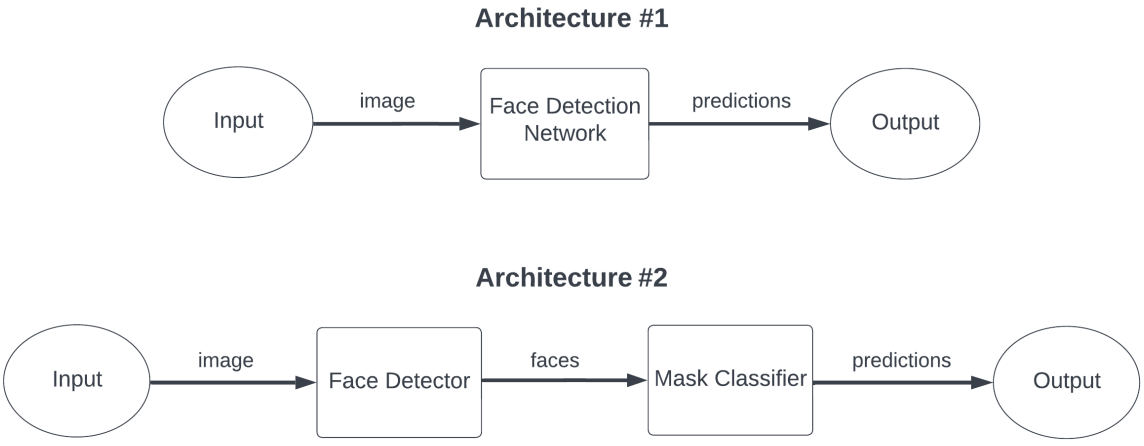

Figure 2.1: The two generic face mask detection architectures.

However, it was also noted that this is dependent on the availability of a larger dataset which would be resource-consuming to curate. This is because this architecture requires a dataset of bounding-box labels including information on face mask use. As a result, pre-trained object detection models for such a specific use-case are lacking.

2.2.3 Architecture #2

On the other hand, the architecture involving a separate face detector and mask classifier has extremely large, publicly available datasets as well as pre-trained models that can be used to optimise it. There are already numerous pre-trained and well-performing face detection models (Ding, Li and Yastremsky, 2021) since face detection has been researched more heavily over many decades. There are also popular pre-trained image classification models (Rath, 2021) that can be tested and adopted by our system.

2.2.4 Choosing an Architecture

Due to these considerations and in an effort to build a complete, high-performing system, the second approach of a separate face detector and mask classifier was used in this project. This will allow for the cherry-picking and utilisation of state-of-the-art, pre-trained face detection and mask classification models.
In the next section, we will use these insights to describe and justify our choice of detection and classification models.

2.3 Detection and Classification Models

2.3.1 Face Detectors

Due to extensive research and implementation in the field of face detection, there are numerous pre-trained face detectors available online (with Code, 2021). In this project, a few popular pre-trained face detectors were used for the development of this system as opposed to reinventing

the wheel by training a custom face detector. This decision better aligned with the scope of the project: The implementation of a complete system which extends the implementation of theoretical models through additional features that can be useful in real-life situations.

Face Detector Criteria

Being able to detect faces efficiently and reliably in a video stream is a core aspect of any face mask detection system.
A successful face detector should satisfy the following criteria:

1. **Fast detections.** Ideally, the detector should run as efficiently as possible. This is especially important since the system needs to perform well in real-time (i.e. process frames as soon as they appear) and with limited computational power (low hardware requirements).
2. **Accurate detections.** The model should be able to detect all faces in an image without any false positives (non-face object detections).
3. **Detection of blurry faces.** Since the visitors may be walking at a fast pace or even jogging into the relevant area.
4. **Detection of angled faces.** In case the camera placement is at an angle to the visitors entering the area.
5. **Detection of diverse faces.** For the identification of all faces regardless of race, gender, and other physical differences.
6. **Detection of faces with accessories (occluded faces).** Since individuals might be walking in wearing glasses, hats, face masks, etc.

Based on these criteria, a set of popular face detectors were chosen for testing and comparison.

Pre-trained Face Detectors

The detectors that were tested and compared in this system are the following:

OpenCV Face Detector[2].
The first detector that was implemented and tested was a pre-trained face detector offered by OpenCV. This is a one-stage detector (SSD) that is based on a ResNet-10 architecture. It was trained for face detection on web images of an undisclosed source ([19, n.d.). This deep neural networks (DNN) detector has been tested against OpenCV's Haar Cascade detector and was shown to be superior in every aspect ([19, n.d.). This detector is a promising candidate since it has also been shown to be effective in face mask detection problems (Kumar, 2021).

Yoloface Detector[3].:
YOLO (Redmon and Farhadi, 2018) and SSD (Liu et al., 2016) are two state-of-the-art object detection tools, each with their own pros and cons (Algoscale, 2021). Since our previous detector was SSD-based, it makes sense to also try a YOLO-based detector. The yoloface detector uses the frequently-used version YOLOv3 (Patil, 2021), and is built for real-time applications, favouring high speed while maintaining high accuracy.

[2] https://github.com/opencv/opencv$_3$*rdparty*/*tree*/*dnn*$_s$*amples*$_f$*ace*$_d$*etector*$_2$0170830
[3] https://pypi.org/project/yoloface/

RetinaFace Detector[4].: The final detector that was thoroughly tested was an adaptation of the RetinaFace detector (Hukkelås, 2021). This is another one-stage (SSD) detector with a MobileNet backbone rather than ResNet. Since the two previous detectors were implemented in TensorFlow and Caffe (Singh, 2021), the RetinaFace detector further increases diversity through its PyTorch implementation[5]. This detector is also optimised for fast computation, performing 6 times as faster on GPU rather than CPU (Hukkelås, 2021).

Other Detectors: Other detectors such as Multi-task Cascaded Convolutional Networks (MTCNN) and Dilb detectors (Rupesh, n.d.) were tested but were too slow on CPU. They achieved 0 accuracy since the system ran at < 1 FPS, and therefore they were not implemented.

Note: There are other detectors that are said to perform better (both in speed and accuracy) such as the face detector from xailient but are not publicly available (Pokhrel, 2019).

Having chosen the desirable face detectors, a similar mindset was used in the second component of our system: face mask classification.

2.3.2 Face Mask Classifiers

Face mask classification refers to the process of analysing an image of a person's face and identifying whether or not the person is wearing a face covering. Similarly to face detection, there has been substantial research and implementation conducted on this problem (Ding, Li and Yastremsky, 2021; Yadav, 2020; Braulio Rios, 2020; Goyal, 2022), especially since the beginning of the Coronavius Disease (COVID-19) in 2020.

Face Mask Classifier Criteria.

The criteria for choosing a face mask classifier are identical to the aforementioned criteria for face detectors.

Mask Classifiers. During real-world testing of our system, it was seen that face mask classification models were much faster and more accurate than face detectors. The initial classifier almost never failed, so we decided to only test 2 distinct models, since they were already performing almost optimally no matter the task. The classifiers that were tested and compared in this system are the following:

1. **MobileNetV2**[6].: The first model used a pre-trained MobileNetV2 model as a feature-extractor, and had its final layers trained an augmented dataset of (initially) $3,833$ images. The model achieved a 91% accuracy on the test set, and the MobileNetV2 architecture is specifically built for computational efficiency (Fransiska, 2019), which is exactly what we are looking for in real-time applications. Finally, the model was tested and proven effective in real-time face mask detection (Srinivas, 2020).
2. **InceptionV3**[7]: Our second and final pre-trained model uses an InceptionV3 model as a feature-extractor, which is also built for fast predictions [45]. The final model achieved a 95% accuracy on the test set and was also proven to perform well in real-time face mask detection (Deb, 2020).

[4]https://pypi.org/project/face-detection/
[5]https://pytorch.org/
[6]https://github.com/balajisrinivas/Face-Mask-Detection
[7]https://github.com/chandrikadeb7/Face-Mask-Detection

When it comes to face mask classification, the number of classes used is an important aspect to consider.

Classes

The simplest implementation of face mask classification involves two classes: mask and no_mask. An additional class of interest would be one for a wrongly worn mask (e.g. covering the mouth and not the nose), for which research implementations exist (Ding, Li and Yastremsky, 2021). However, since most high-performing, pre-trained mask classifiers can only predict two classes, we chose for the system to only predict mask and no_mask. An extension of the system, given more time and resources, should include the third class for incorrectly masked people. Using the basic classes however, we added an "uncertain" output when the probabilities for mask and no_mask are near equal. This functionality can be tuned by the user to filter out unreliable predictions made by the model, which oftentimes includes wrongly masked individuals.

Next, we will discuss another algorithmic addition to our system that will provide us with functionality that is rare in face mask detection systems.

2.4 Object Tracking

2.4.1 Introduction

One important process that is not always used in face mask detection implementations is *object tracking*. Object tracking can provide benefits such as improved classification accuracy, data collection, and counting functionality. Tracking becomes relevant when there are multiple related images (such as frames in a video stream) on which object detection is performed. It is the process of assigning a unique ID to each object that is output by the detector, and tracking the unique objects as they move through frames while storing the relevant information (Meel, 2021). When considering whether such functionality is beneficial in our use-case, we should closely examine the advantages and disadvantages of object tracking.

2.4.2 Disadvantages of Object Tracking

Tracking has a few disadvantages that should be considered in a real-time system such as ours.

1. **Computational complexity.** One core disadvantage of object tracking is increased computational cost (Meel, 2021). This can make the real-time system run significantly slower, resulting in a low frame-rate (which in turn leads to less accurate results). However, methods to mitigate this were taken into account.
2. **System Complexity.** An additional consideration is that it increases the complexity of the system, making it more difficult to debug and maintain.
3. **Tracking sensitivity.** Finally, tracking can be sensitive to lighting, background distractions, changing objects, disappearing objects, and occlusion (merging objects) (Meel, 2021) all of which can lead to malfunctions and inaccurate results.

That being said, there are key advantages to object tracking (Braulio Rios, 2020) that make it an imporant addition to our system.

2.4.3 Advantages of Object Tracking

The advantages most relevant to our use-case are the following:

1. **More Accurate Predictions.** Tracking allows our face mask classifier to process the same face multiple times and then average the results for a more accurate prediction.
2. **People Counting.** Identifying unique faces and tracking their trajectories in a video stream provides as with a reliable way to count individuals entering an area, which is the second main goal of our system. Without object tracking this would not be possible.
3. **Statistics Collection.** Finally, identifying unique faces allows us to also catalogue their appearance and export useful statistics on face mask use (such as the percentage of masked people encountered) that can be used for insights at a future time. No known implementations of a similar system have included such a feature in scientific literature (Ding, Li and Yastremsky, 2021; Yadav, 2020; Goyal, 2022), though ML engineers in Tryolabs claim to have done so (Braulio Rios, 2020).

Due to these key advantages, it was decided that tracking is an essential addition to our system.

In the next few paragraphs, we will mention key differences between object tracking algorithms along with the implementation choices that were used for our system.

2.4.4 Single Object Tracking vs Multiple Object Tracking

Single Object Tracking (SOT) refers to the tracking of a single object. This is usually detection-free tracking (Barla, 2022) as the bounding box of any object is manually provided to the tracker (many OpenCV trackers work this way (Rosebrock, 2018). In our application, we are interested in *Multiple Object Tracking* (MOT) where the algorithm keeps track of multiple objects in each frame. This is because there may often be multiple masked and unmasked people in a given video stream.

2.4.5 Batch Method vs Online Method

When it comes to the information available to the tracker, there are two categories of tracking: Batch and Online (Meel, 2021). Batch tracking algorithms use information from future frames to perform object tracking, while online tracking algorithms only use past and present information. Batch tracking algorithms generally outperform online tracking ones (Meel, 2021), but in our case are infeasible since are system runs in real-time. Thus, our implementation uses the online tracking method.

2.4.6 Visual Tracking vs Tracking-By-Detection

Every tracking system requires at least one step of object detection. After one or more objects have been detected, tracking ensues.

Visual Tracking

In visual object tracking, the object detector is run only once (at the beginning of a video sequence), and the tracking algorithm uses the initial state of the detected objects (such as their centre location and scale) to estimate its future positions and track it throughout the video sequence using advanced mathematical algorithms (Xu et al., 2019). One major advantage of visual tracking is that, since it runs the detection step only once, it is very computationally efficient.

Tracking-By-Detection (TBD)

On the other hand, in tracking-by-detection (TBD), object detection is performed in every single frame of the video. In this way, objects detected in new frames are matched to the previous objects by simple metrics such as the Euclidean distance. TBD algorithms are older and much simpler than visual tracking ones (Bochinski, Eiselein and Sikora, 2017), but can be highly effective too.

Choosing between the two

1. **Detection Frequency.** For the purposes of our system, visual tracking can be advantageous due to its computational efficiency resulting from fewer detections. However, our system needs frequent detections since new visitors might appear in the frame at any time, and existing visitors may change their face mask status as they are entering the area, which requires an instant update.
2. **Detection Reliability.** In visual object tracking, one obstacles to overcome is object variation (Meel, 2021). The visual tracker should be able to keep track of an object regardless of changes in its size and characteristics, which can often be difficult. In TBD, this task is handled more successfully by deferring it to higher-performing object detectors, since the object is re-detected in every frame.
3. **Simplicity.** Visual tracking algorithms can be extremely complex in an effort to infer future object positions (Xu et al., 2019). This, In addition to making them run slower than their TBD counterparts, it makes them less intuitive and more difficult to debug.
4. **Customisability.** Finally, due to their simplicity, tracking-by-detection algorithms can be tuned and customised for specific use-cases much more easily, making them more diverse and hence a favourable option for this project.

Unfortunately for TBD, most pre-built algorithms use visual tracking, including OpenCV's built-in tracking algorithms (Rosebrock, 2018). Thus, for this system, a custom tracking-by-detection algorithm was implemented which is based on *centroid tracking* (Khandelwal, 2022). Our custom tracking library simply tracks faces by calculating their Euclidean distances in subsequent frames and using it to associate them. The added customisability of a custom tracker gave us the opportunity to implement people counting and other useful functionality within our tracking library. Moreover, tracker parameters can be modified to fit different use-cases. We will go into the implementation details of this library in a later chapter.

In the next chapter, we will discuss the second main task of our system which object tracking enables: *People Counting*.

2.5 People Counting

As mentioned in the introduction, people counting refers to the manual or automated counting of individuals traversing a given passage or entrance (e.g. a shop or a public space). We will start by differentiating between counting types and technologies, before concluding on the methodologies that were chosen for our system.

2.5.1 People counting vs Crowd Counting

Crowd counting (special case of object counting) is a technique used to count or estimate the number of people in an image (Khandelwal, 2021). While it shares many similarities with people counting, its main focus is to estimate the number of people that already exist in an area at a given time, instead of counting the people that enter and exit the area in real-time. As implied, people counting introduces the idea of *human flow control*, where the direction in which the people are moving is also taken into account. This functionality allows for the task of counting people entering or leaving the area. Note that counting in our system does involve human flow control, but only counts the people that enter (and not exit) a given area.

2.5.2 People Counting Technologies

Unlike face mask detection that saw a recent spike in popularity, automatic people counting systems have been consistently researched and applied since the 1990s (Metro, 2022).

(1990s) Manual clickers. In the 1990s, up to which point manual clickers were used to count people, we saw the implementation of pressure-sensitive mats that determined whether a new person entered a space by the force of their weight on the mat.
(2002) Infrared Beam Counters. In early 2002, there was an increase in popularity of infrared beam counters (Metro, 2022) which work by detecting the interruption of an infrared light signal when an object passes through.
(2005) Thermal Counters. Beam counters struggled with unusual behaviour such continuous disruptions done by the same individual, and this was partially fixed in 2005 by thermal people counters that analysed the heat distribution fluctuations when people entered an area (FootfallCam, 2017).
(2011) WiFi- and Camera-based Counters. In 2011, Wi-Fi technology was utilised for people counting through the use of unique Wi-Fi request signals from smartphones to identify customer movement. During the same period, the most modern people counting methods were utilised which involve the processing of images using computer algorithms. These camera based methods are most relevant to the system that was developed for this project.
(2018) AI-Powered People Counters. The next generation of people counters leverage advanced computer vision algorithms and machine learning to analyse video data and extract a people count. These methods became possible with the increased availability of powerful computer hardware and the rising success of deep learning.

Our System

Our system uses an AI-Powered people counter to test the technology and because the functionality is a direct consequence of statistics-orientated face mask detection.

2.5.3 Two Camera-Based Counting Methods

When it comes to AI vision counters, there are two main ways to count objects.

1. Unique Object Count.

In this method, as soon as a "new" object appears it is immediately added to the total object count. An object (e.g. a face) is seen as "new" if the tracker has no records of it. This method of object counting requires highly accurate object detection (no false positives), as well as reliable tracking to make sure that an object is not lost and re-detected, which can lead to counting it twice. It has been used successfully in many counting implementations (Braulio Rios, 2020; Retail, 2022; Padmashini, Manjusha and Parameswaran, 2018).

2. Boundary Line Object Count.

Another way to count objects is by drawing a digital boundary line, and counting only the objects that cross that line (see 2.2.

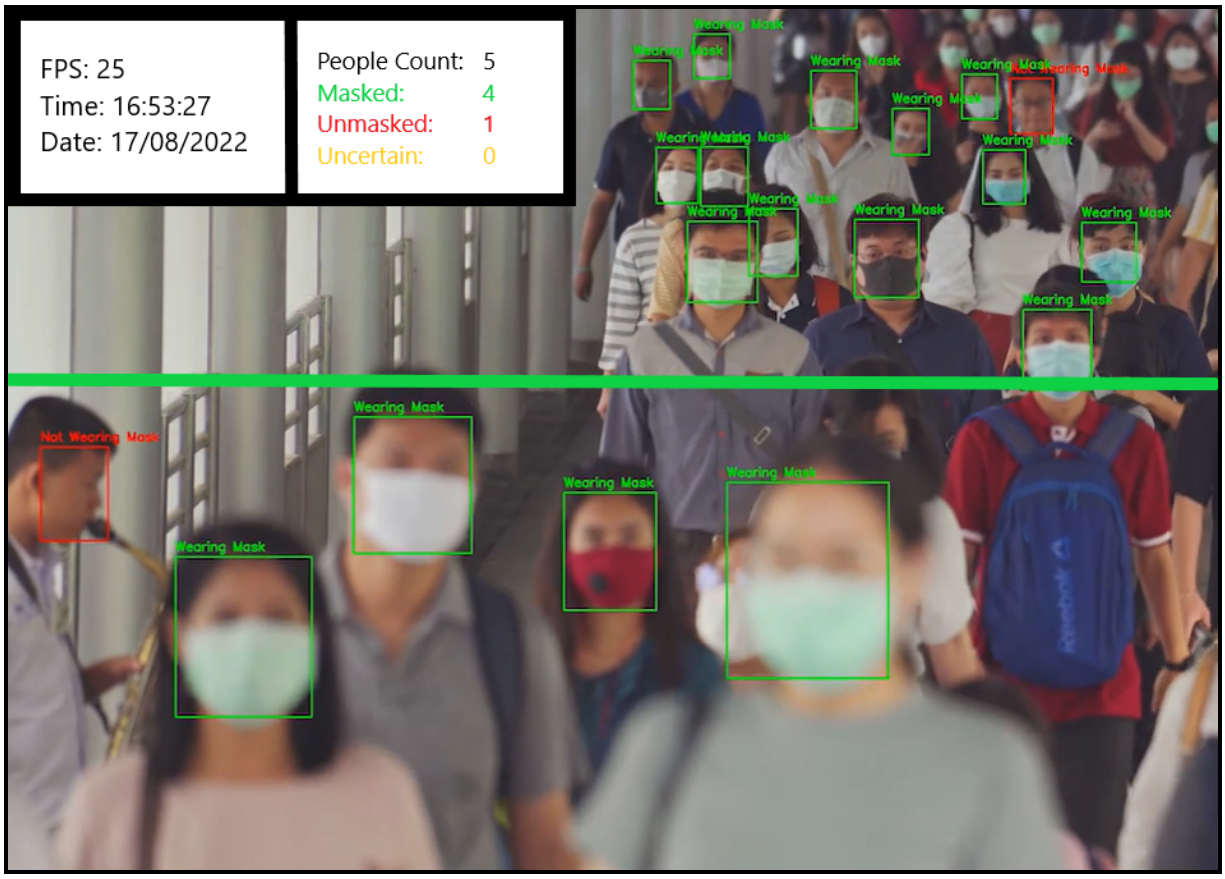

Figure 2.2: Example of our system's output. The horizontal, green line (boundary line) acts as a counter each time a face crosses it. The counts along with the face mask predictions are displayed in the top left corner along with other relevant information. (Edited image of Karchaudhuri (2020)).

In this method, the counter checks three conditions:

1. If an object has appeared before the boundary line.

2. If the object is positioned after the boundary line.
3. If the object has not been counted before.

If all three conditions are met, then the counter increases by 1. While this method requires appropriate placement of the boundary line (this is a parameter), it is a popular method in people counting implementations (Canu, 2021; Zhenqiu Xiao and Tjahjadi, n.d.) because it easily detects whether a person is entering or exiting an area.

Choosing a Method

For our system, choosing the right method was not difficult. Having in mind that the camera will be placed (probably) at an entrance to an area with the goal of counting how many people enter that area (e.g. a shop), the second method already seems more beneficial. In addition, the boundary line method is less susceptible to the type of overcounting that may happen when an object briefly disappears for a few frames, and therefore does not require a robust detector-tracker to achieve high accuracy. One final consideration is that the second method allows for more accurate face mask detection by allowing for averaging or classification predictions. While a face is before the line, the classifier's predictions are accumulated and then averaged as soon as the face crosses the line and their count and final label are catalogued.

Thus, due to these significant factors, the boundary line method was implemented in our system.
People counting is the final aspect of our system and this concludes the background chapter. Next, we will discuss the requirements specification for our system which is based

Having described the final aspect of our system in people counting, we are ready to use the information in this chapter to discuss the requirements specification for our system.

Chapter 3

Requirements Specification

3.1 Introduction

This section outlines the user and system requirements for the Real-Time Face Mask Detection & People Counting System implemented as part of this project. The intended readers are users or current and future developers of the software system.

3.2 Overview

In today's world, where most retail businesses and government organisations have access to basic computer hardware and a data analytics team[1], automated data collection can be of great benefit. This is especially true when the data can provide commercial benefit while making it easier to abide by government regulations and protect the safety of a company's visitors. The Real-Time Face Mask Detection & People Counting System aims to satisfy these requirements.

In terms of the application scope, this software is intended to benefit individual users and small business owners, as well as large commercial business and organisations. In principle, only minimal technical knowledge is required to use it effectively. However, since this is a prototype, users should be prepared to be expected to troubleshoot technical issues which might require the advice of an expert.

3.3 User Requirements

3.3.1 Real-time Processing

The intended use case of the system is to process a live video feed (e.g. produced by camera outside of a store) to detect people passing by an area or entering a closed space. Therefore, the system needs to be able to detect and process faces in real time. In addition, it should provide immediate feedback (via a user interface) about the detections it has made from the moment it was turned on.

[1] https://financesonline.com/relevant-analytics-statistics/

3.3.2 Face Detection & Classification

The system should detect faces as they enter the live video feed and identify whether the individual in question is wearing a face covering or not. It should also track faces as they move in-frame and provide real-time visual feedback about the faces it has detected and the state they're currently in (e.g. masked or unmasked).

3.3.3 People Counting

The system is expected to be used to track people entering a particular area and provide statistics to enable identification of patterns. Therefore, the system should define a boundary in the video feed, such that an observation about whether a person is wearing a face covering properly or not is recorded every time a person crosses that boundary (the boundary can be, for example, the entrance to the building that the system is set up for). This information should be displayed both on the user interface as real time counts of the total number of people, the total number of masked people, and the total number of unmasked people that have crossed the boundary.

3.3.4 Data Collection

In addition to displaying information on the user interface, the system should collect and export data for further analysis. Specifically, a dataset of all observed instances of masked and unmasked people that have entered the specified area should be exported to permanent storage along with a timestamp field.

3.3.5 User Classes and Characteristics

3.4 System Requirements

System execution is subject to a set of requirements that a host machine needs to meet in order to robustly run the built application with adequate performance. These were determined through system testing. The specific requirements, outlined in the sections below, are largely a result of limitations with respect to time constraints and the the environment and technologies used during development. For example, the application was built for Microsoft Windows because that was the only available platform at the time of development.

3.4.1 Standard Usage

The following system requirements apply to the standard usage of the application, wherein a user executes the program for normal operation.

- Storage space (required for installation): 1GB
- Operating systems: Windows 10
- Minimum Hardware specifications: CPU: AMD Ryzen 5 2500x (or better), Webcam: 400x400 pixels
- Minimum frame rate: 8 FPS
- Language: English

3.4.2 Development & Debugging

For development, there are additional system requirements that depend on the technologies used to develop the application.

- Software: Python $>=$ 3.9
- Operating systems: Windows 10 (only for running the built executable)

3.5 Design and Implementation Constrains

3.5.1 Hardware Constraints

The system requires the previously stated software and hardware to function as intended. If the minimum hardware specifications are not met, then the software would become less reliable in collecting accurate data and could even crash mid-use.

3.5.2 GDPR Laws (UK)

Since the collected data is anonymised, it is not subject to the restrictions placed under the General Data Protection Regulation (GDPR) [2].

[2] https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/what-is-personal-data/what-is-personal-data/

Chapter 4

Overall design

4.1 System Architecture

The system is structured as a series of components which progressively receive, process, and propagate data through the system (Fig. 4.1). There is a single input source, a video camera, which produces a data stream, from which frames are read one at a time. The system processes each frame and updates the user interface accordingly. This is a synchronous, iterative process that repeats for each new frame until the program is terminated. Additionally, the processed tracking data is periodically exported to persistent storage for future use (e.g. offline analysis).

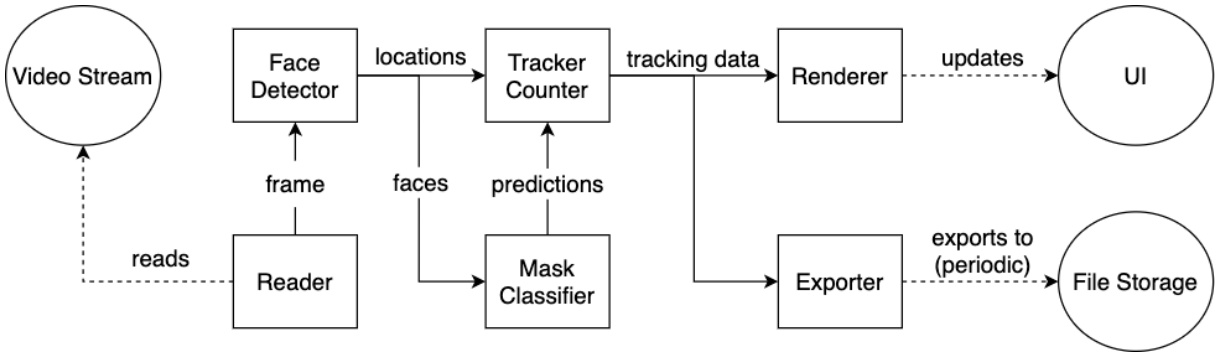

Figure 4.1: System Architecture

4.1.1 Input

The Video Stream represents the raw video data recorded by a physical device (camera).

The Stream Reader is an auxiliary component which consumes raw frame data from the Video Stream, one frame at a time, and outputs a high-level representation of each frame, to be processed by the rest of the system.

4.1.2 Detection & Classification

The Face Detector receives a frame from the Stream Reader as input, which it processes to identify all human faces that are visible in the frame. It produces two outputs for each face

it detects: a data representation of the face and the location in the frame at which it was detected.

The Mask Classifier receives faces identified by the Face Detector and classifies them into one of three categories: mask, for faces wearing a mask, no-mask, for faces not wearing a mask, and uncertain, for cases in which the classifier failed to make a decisive prediction. It then outputs those predictions.

4.1.3 Tracking & Counting

The Tracker Counter is a stateful component that maintains the set of locations of all faces detected in the video stream, as well as counters for all unique people detected throughout the execution of the system. It accepts two inputs: the set of locations on the current frame at which faces were detected by the Face Detector, and the predictions produced by the Mask Classifier for each face. It uses that information to (1) update its internal state and (2) output updated tracking information to be used as the system's output for the current frame.

4.1.4 Output

User Interface

The Stream Renderer is an auxiliary component which uses the object tracking data from the Tracker Counter to update the user interface. This provides the visual representation of the face-mask detection information produced by the system, which constitutes the first output of the system.

Persistent Storage

The second and final output of the system is produced by another auxiliary component, the Exporter, which periodically persists the tracking data produced by the Tracker Counter component to permanent storage for offline analysis.

Chapter 5

Detailed design and implementation

The system is implemented in Python, which is a very suitable language for this use case due to the available libraries (e.g. NumPy[1] and pandas[2]) that provide abstractions for representing and performing operations on tabular data. It also enables efficient loading and execution of Machine Learning models via TensorFlow[3] and Keras[4], which are used for the detection and classification parts of the system. The main system execution loop performs each operation outlined in the system diagram sequentially for each frame in the video stream. The rest of this chapter discusses important design and implementation details of the main components of the system. The full code can be found in Appendix C.

5.1 System Configuration

The system is configurable via a number of parameters that affect its performance. In order to pick optimal values for this implementation of the system, parameter tuning was performed to test different combinations of parameter values and evaluate the resulting performance. The full list of parameters is available in Appendix C along with the chosen value for each one.

5.2 Input

The Python imutils[5] library (which uses OpenCV[6] under the hood) is used to read byte data from a video recording device and transform it into a numpy ndarray[7] object. Each array represents a single frame in the video as an image. The library can also be used to process video from a file, which is useful for testing the system with pre-recorded footage.

[1] https://numpy.org/
[2] https://pandas.pydata.org/
[3] https://www.tensorflow.org/
[4] https://keras.io/
[5] https://pypi.org/project/imutils/
[6] https://opencv.org/releases/
[7] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html

5.3 Detection & Classification

The Face Detector uses a pre-trained face-detection model which accepts an image as input, which potentially contains many faces. It outputs a 2-D array in which every row corresponds to a detected face and contains the coordinates of the bounding box surrounding the face in the frame. These coordinates are used by the Tracker Counter component to track the location of people detected in the video feed. A number of candidate models have been tested and compared based on accuracy (discussed in Chapter 7).

5.4 Tracking & Counting

As mentioned in Chapter 4, the Tracker Counter is a stateful component which performs object tracking to identify unique faces (people) seen on the input video stream and record statistics about them. It is implemented as a singleton class, which is initialised once and reused throughout the life-cycle of the system. The main parts of the class are a set of state variables and a method which updates the state at each frame.

5.4.1 Component State

The state of the component consists of 4 counter variables and 1 dictionary with tracking information. The counter variables record the total number of people detected by the system, and the number of people labeled as masked, unmasked, and uncertain by the Mask Classifier. Each entry in the dictionary represents a single person and contains a unique ID (key) and tracking information (value), such as the location of the person in the current frame (represented by the centroid of the bounding box computed by the Face Detector) and the history of predictions made for this person by the Mask Classifier.

5.4.2 Component Update

The component includes an update method which is called once per frame to update the component state. It updates the dictionary of tracking information by adding an entry for each newly detected face, updating the coordinates of existing entries, and removing entries for faces that have moved out-of-frame. Determining whether a given face has been seen before or not is achieved by a simple algorithm provided in pseudocode in Listing 5.1. If the given face is sufficiently close to one of the faces in the previous frame it's considered to be the same face at an updated position. This is true unless all faces in the current frame have already been matched up with faces in the previous frame, in which case the current face must be new.

Listing 5.1: Face-tracking algorithm

```
num_faces_matched = 0
for cur_face in faces_in_current_frame:
  face_match_found = false
  for prev_face in faces_in_previous_frame:
    if dist(cur_face, prev_face) < min_dist and num_faces_matched <
       len(faces_in_previous_frame):
      prev_face.position = cur_face.position
      face_match_found = true
      break
  if not face_match_found:
```

```
add_new_face(cur_face)
```

In order to update the counter variables that keep track of the number of masked, unmasked, and uncertain people that have been detected, the update method checks the location of each face in the current frame to determine if any of them have crossed the pre-determined boundary line on the video stream. An interesting implementation detail that is worth mentioning here is that the Tracker Counter receives a new prediction from the Mask Classifier at every frame. And since the label of a face can change (e.g. from masked to unmasked) from the time it enters the video stream until it crosses the boundary line (e.g. by the person pulling down their mask before they cross) the update method computes an average over the predictions recorded in the component state over the last few frames to determine the final classification label.

Finally, the update method returns the coordinates of each face in the current frame, which will be used to produce the system's output.

5.5 Output

As discussed previously, the system displays visual information in a Graphical User Interface (GUI) and periodically exports data to persistent storage for further analysis.

5.5.1 GUI

As soon as the system is executed it launches a window with a GUI which displays the video stream in real time with visual information overlaid on top (you can see two example snapshots in figure 5.1).

The data that are used to generate the visualisations are provided by the Tracker Counter component. Boundary boxes are drawn around each detected face at every frame and the people counting statistics displayed at the top of the GUI window are updated every time a face crosses the boundary line (shown in green in the example in Fig. 5.1).

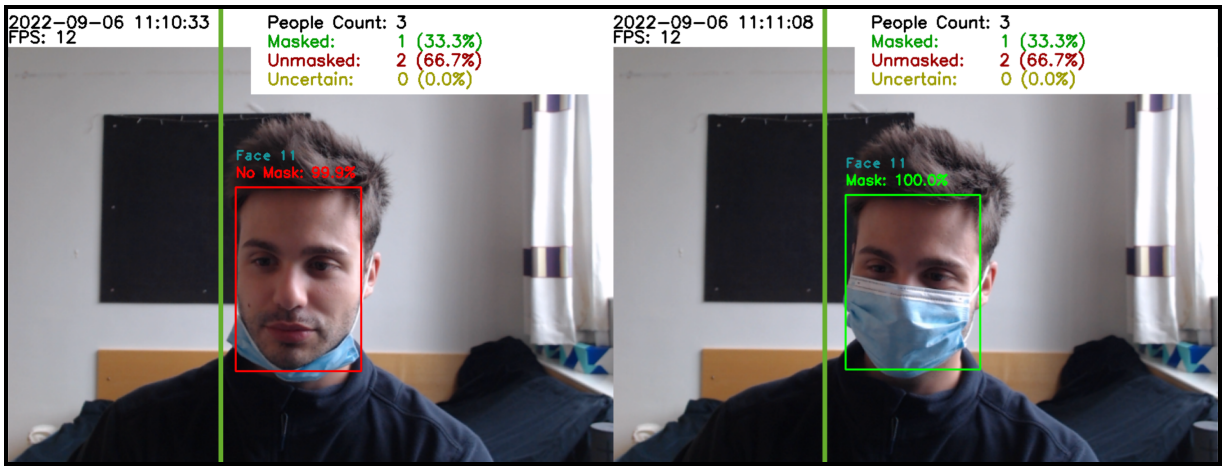

Figure 5.1: Example of the system's GUI.

5.5.2 Persistent Storage

As the final part of its output, the system periodically exports data for further analysis. Specifically, it exports a csv file which lists all classifications made along with the classification timestamp (the time at which each of the classified faces crossed the boundary line). An example of such a csv file is provided in Fig. 5.2. Additionally, the system generates an image (png) of a pie chart which visualises the proportions of faces classified as masked, unmasked, and uncertain (Fig. 5.3).

looks good g, submit.

	A	B
1	Label	Datetime
2	no_mask	05/09/2022 16:33
3	mask	05/09/2022 16:33
4	mask	05/09/2022 16:34
5	mask	05/09/2022 16:34
6	no_mask	05/09/2022 16:34
7	mask	05/09/2022 16:34
8	no_mask	05/09/2022 16:35
9	no_mask	05/09/2022 16:35
10	mask	05/09/2022 16:35
11	mask	05/09/2022 16:36
12	no_mask	05/09/2022 16:36
13	mask	05/09/2022 16:36
14	mask	05/09/2022 16:37
15	no_mask	05/09/2022 16:37
16	mask	05/09/2022 16:37
17	mask	05/09/2022 16:38

Figure 5.2: The contents of an example csv file exported periodically by the system.

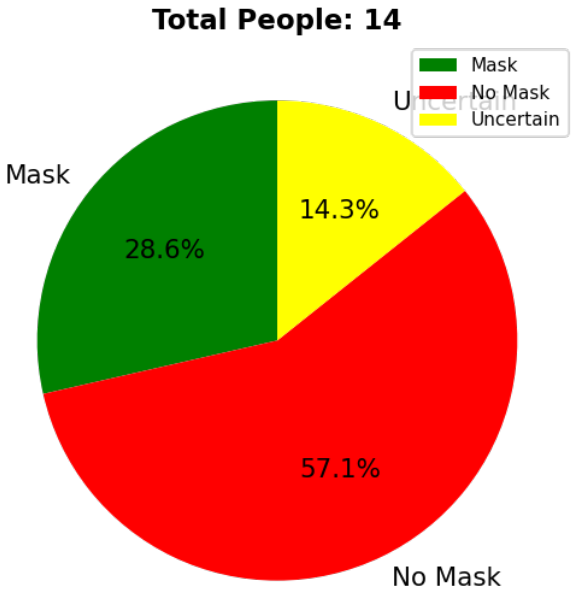

Figure 5.3: Example pie chart generated periodically by the system.

Chapter 6

System testing & Experimental Design

6.1 Introduction

Now that we have described the implementation details of our system, we will see how the system was tested to ensure that it functions as intended. Since this project includes an experimental component in addition to the software development one, a description of the experiments that were done will be mentioned too.

Testing Strategy
In real-time face mask detection it is important that a system can be installed and work as intended for any type of user. A failure to do so can result in unreliable and misleading long-term data. Thus, at first, every single component of our system was tested individually, before using the system as a whole in a real-life application.

Note: In the course of this chapter, we will also perform a critical evaluation of the testing & experimental process.

6.2 Manual Testing

Manual testing, as the term suggests, refers to a test process in which a developer manually tests the software application in order to identify bugs. This differs from *automated testing* where software tools are applied to automate the process of validating a software product (Unadkat, 2021). In our case, both manual and automated testing were performed. We will start the discussion with manual unit testing.

6.2.1 Unit Testing

Unit Testing involves verification of individual components or units of source code (e.g. functions) (Unadkat, 2021) and it is an essential part of any software development project. Throughout the project, extensive unit testing was manually performed. Some of the test cases that were validated include the following:

- The detector should detect faces and output corresponding bounding boxes for each one.
- The classifier should correctly process each face provided by the detector and produce a single output for each.
- The counter should function as expected in a variety of scenarios in which an object crosses the boundary line.
- The system should display information correctly on the GUI and periodically export data in the appropriate format, on the specified interval.

This also included extensive parameter tuning for parameters such as the minimum distance in Euclidean tracking, the placement of the boundary line for people counting, the height/angle of camera placement and more.
After it was confirmed that system units function as intended, we moved onto integration testing (Unadkat, 2021) which tests the functionality of multiple units.

6.2.2 Integration Testing

In integration testing, the core components of the system (the Face Detector, Mask Classifier, and Tracker Counter) were tested together to see if the intended result was obtained. The results were verified by observing the visual and numerical output on the video stream.

Finally, we moved on to automated *System Testing*, where the entire system was tested end-to-end.

6.3 Automated Testing & Experimental Design

To achieve automated testing, separate code was written to execute the whole system through a series of test cases using pre-recorded video files. As part of the same process, the video files were used for the experimental comparison of the face detectors and mask classifiers mentioned in Chapter 2.

6.3.1 Video Setup

The most important aspect of our system is that it can be useful for a company or an organisation in a real-world scenario. Thus, it was decided that the best way to test our face mask detection and people counting system would be to replicate a real-world situation of people entering an area to ensure intended functionality and assess performance.
Five videos were filmed by myself to be processed by our system. It is important that video files were used instead of a real-time test to ensure consistency across all tests.

Video Setup

Participants and Characteristics: The videos involved 4 unique individuals: 2 males (in their 20s) and 1 male and 1 female (in their 50s). The skin colour varied from white to brown, and hair was non-existent (bald), short, or long (covering the forehead with bangs).
Facial Accessories: The masks that were worn were typical light blue surgical masks, though the system was separately tested with masks of varying types and colours (+ sunglasses and a hat) and there was no noticeable difference in system performance.

Camera Type: The participants were walking by a phone camera (frame width x frame height: 1080x1920 pixels, FPS: 30). The camera specifications were chosen to align with in-store camera capabilities (Watch, 2019; Mesnik, 2019).
Camera Setup: The camera was placed at a height of 1.70 meters at an angle of 15 degrees from the line that participants entered the area (so there was a default side-face element in detection, i.e. participants were not facing the camera). Note: This setup was shown to be more successful at face detection than the camera being placed at an angle of 0 but a height of more than 2.0 metres.
Lighting and background: The environment in the videos had average brightness and natural light (closed space). The background was relatively complex (glass door with light reflection, light-skin-coloured wall, covered piano) though this did not appear to affect detector performance.

6.3.2 Video Descriptions

To ensure that system testing covered a variety of real-world applications, 5 videos were filmed that were each about 1 minute in length. Each video covered a different aspect of the real world which allows us to see how each model that was used performs in each, as well as the system as a whole. The videos that were filmed were the following:

1. Baseline Video.
The purpose of the first video was to reflect a simple and standard use-case of our system. In the video, individuals enter an area one at a time, facing forwards, and walking at a regular pace. In total, 8 masked people and 8 non-masked people (combination of 4 unique individuals) walked in the room. The real-world analogy of the video is to set up a camera at the entrance of an area (e.g. a store) where people walk in individually, without unforeseen movements.
2. Speed Video.
The second video is the same as the first one with the exception that the individuals are walking fast or running by the camera. This is something that can happen in a real-world scenario if a customer is in a hurry or trips as they enter the store, and reflects the system's ability of detecting blurry faces.
3. Sideface Video.
In a similar way, the "sideface" video is a replica of the first video with the individuals having their faces stare away from the camera (at a 90 degree angle) so that only the side of their face shows. This may happen in the reality as a person gets distracted as they are entering the store. In addition, it reveals how the system performs with sub-optimal camera placement (e.g. due to store design).
4. Multi-person Video.
The fourth video is the same video as the first one but with individuals entering the area in side-by-side pairs (6 masked and 10 non-masked in total, including combinations of masked and non-masked). This shows how well the system can handle detections of multiple objects (more than one face) which require increased computational time. In addition, it shows how the system performs when an individual's face might be obstructed by another person.
5. Chaos Video.
Finally, the "chaos" video involves a chaotic scenario which presents all of the challenges previously mentioned and more. In this video, multiple individuals enter the area facing in any direction and at any pace. Individuals are simultaneously exiting the area or standing by the camera. In addition, couples are holding each other and individuals may be putting on or removing their mask as they are crossing the boundary line.

This video reflects a challenging scenario in which zero external measures have been taken to assist the system in accurate detection (such measures could be requiring visitors to enter one at a time or face the camera at all times). Nevertheless, it is useful to see how the system performs in a near worst-case scenario.

Next, we will describe the hardware on which the system was tested.

6.3.3 Testing Hardware

The system was run on a personal computer of the following (relevant) specifications:

- CPU: AMD Ryzen 7 3700X
- RAM: 32GB

It should be noted that the system was run on a CPU rather than a GPU (due to GPU incompatibility), which made it significantly more slowly. While frame rate differences were eliminated to ensure a fair detector comparison (FPS=30), the average frames per second (FPS) that each detector could process (which is relevant in a real-time application) were still calculated and will be compared in the next section. Along with this critique and before discussing the results, we will critically evaluate our testing process.

6.4 Critical Evaluation

Even though all components of the system were tested, the process can be expanded upon and improved given fewer computational and ethical limitations and more time. Specifically, unit testing should be automated by writing functions that run them automatically. This can save time in the long-run and make the system more reliable. In addition, the sample size and sample diversity should increase significantly in the experiments/system testing, which could not be done in our case due to ethical considerations and limited time. A highly beneficial testing method would be to provide the system to a company and let them use it for a few days, provide ease-of-use feedback, and assess the accuracy of the results in their use-case.
Now that we have outlined the testing/experimental process, we move on to the results that we obtained from applying our system to the video files.

Chapter 7

Results

In this section, we will discuss how the system performed on the aforementioned video files.

7.1 Testing Results.

Regarding the system testing component of the experiment, the system functioned as expected in all situations, passing the most important tests. This shows that the system can be applied without issues in a similar real-world environment and be expected to function as intended. In the next section we will discuss the experimental side of the project, where we compare detector and classifier performance in terms of counting and classification accuracy.

7.2 Detector Comparison

The task of the detector is to reliably localise all faces in each frame, allowing the classifier to label them as mask/no-mask and the tracker/counter to track them and document the results. Thus, to assess detector performance, we looked at the percentage of people that entered the store and were successfully counted (accuracy). We further break down this percentage by whether the individuals were masked or non-masked, which gives us an indication of the detector's ability to detect each class. Results for all three detectors are summarised in the tables below.

Note: These accuracies depend entirely on the detector because the classifier, tracker, and counter functioned precisely as intended. Thus, these metrics reflect detector performance exclusively.

1. OpenCV Detector (12FPS, SSD, ResNet, Caffe)			
Test	Total Count	Mask Count	No-Mask Count
Baseline	100%	100%	100%
Speed	100%	100%	100%
Sideface	75%	63%	88%
Multiperson	94%	83%	100%
Chaos	67%	64%	75%

2. Yoloface (9FPS, YOLOv3, TensorFlow)			
Test	Total Count	Mask Count	No-Mask Count
Baseline	69%	38%	100%
Speed	56%	13%	100%
Sideface	31%	0%	63%
Multiperson	69%	17%	100%
Chaos	53%	45%	75%

3. RetinaFace (7FPS, SSD, MobileNet, PyTorch)			
Test	Total Count	Mask Count	No-Mask Count
Baseline	100%	100%	100%
Speed	88%	75%	100%
Sideface	88%	75%	100%
Multiperson	94%	83%	100%
Chaos	80%	82%	75%

Comments

1. OpenCV Detector:
   - Excellent performance on regular and blurry (fast-moving) faces.
   - Very good performance on multiple faces (especially if they are not masked).
   - Mediocre performance on sidefaces and a chaotic scenario.
   - Fastest detector on CPU (12FPS).
2. Yoloface Detector:
   - Mediocre-low performance across the board (especially on masked faces).
   - Relatively slow on CPU (9FPS).

3. RetinaFace Detector:

- Excellent performance on regular faces.
- Very good performance on blurry (fast-moving), side, and multiple faces (especially if they are not masked).
- Good performance in a chaotic scenario.
- Lowest FPS on CPU (7FPS), though as mentioned in chapter 2, it can run x6 faster on GPU.

Choosing a Detector

The Yoloface detector is a good example where the detector was probably not trained on masked faces and hence had a very difficult time detecting them. Thus, it is not ideal for this use-case. The choice comes between the built-in OpenCV and the RetinaFace detectors. While the RetinaFace detector performed better in the sideface and chaotic scenarios, it was outperformed in the speed one. Ultimately, this comes down to the intended application decided by the user, but for our system, the OpenCV detector was chosen when concluding on the recommended use-case of our system which would be a controlled, baseline one.

7.3 Classifier Comparison

During testing, it was seen that the classifiers were functioning as intended (for the most part). In the vast majority of the system's failures, only a tiny proportion was due to misclassification of a face. Instead, most of them were the fault of the detector. In addition, the computational cost of mask classification was negligible when compared to that of face detection.

In total, there were 6 detector-classifier combinations that were tested, each on all 5 videos, and the classifiers had 0 false positives and 0 false negatives in all tests, which is optimal for reliable data. The accuracy for both classifiers across all tests was 95% because of a few "uncertain" predictions on only masked individuals, but this could possibly be tuned by decreasing the required confidence for classification (i.e. the "uncertain interval"). In addition, none of the "uncertain" classifications were were in the baseline test, which is the recommended use-case for this system. In the end, since both classifiers performed very similarly in accuracy and computational cost, the first MobileNetV2[1] classifier was chosen since its unit testing was more thorough as it was the first classifier of choice.

In the next and final section, we will draw conclusions and report ways in which the system can be improved in future work.

[1] https://github.com/balajisrinivas/Face-Mask-Detection

Chapter 8

Conclusions

As demonstrated from the elaborate testing and experimental results, the system is capable of both people counting and face mask detection, with reliable data collection. The recommended use-case of the system is a baseline scenario where visitors should enter the store one at a time, and it should be ensured that their face is clearly visible. The system can still function in a more complicated scenario, but lower accuracy should be expected. In theory, the system is ready for adoption from companies, though there should (ideally) be supervision of its use from a data scientist or a machine learning engineer.

Future Work

In terms of improving the system, there are numerous suggestions:

- **Computational Efficiency and Testing:** The source code can be further optimised for the system to run faster. In addition, more automated testing & experiments can make the system more reliable.
- **Camera Compatibility:** Make the system compatible with more cameras that are commonly used in retail (Watch, 2019). IP Camera compatibility is trivial to implement (by changing one argument) (Employee, n.d.).
- **Improved GUI:** The graphical user interface could be heavily optimised for clarity and ease-of-use, see (Meraki, 2022,?; Eocortex, 2022).
- **Head Detection:** Instead of a face detector, head or human body detectors can be utilised instead. These can lead to more reliable tracking and counting, though they might also slow down the system.
- **Exit Count:** With the use of a head detector/pose estimator (Braulio Rios, 2020), people facing away from the camera could also be tracked and counted.
- **Automated Emails:** The system could sent email notifications as an alert when a non-masked individual enters the area.

That being said, the system achieved the minimum requirements and, as originally aspired, can satisfactorily perform automated face mask detection and people counting for the average consumer. This is significant because it further confirms that years of computer vision research insights & model implementations can be directly adopted and used in a simple, semi-custom

system to automate everyday tasks. While these tasks provide monetary benefit for the developers, they also contribute to public health and overall societal efficiency.

Bibliography

n.d. [Online].

Algoscale, 2021. *Yolo vs ssd: Which one is a superior algorithm* [Online]. Available from: https://algoscale.com/blog/yolo-vs-ssd-which-one-is-a-superior-algorithm/.

Andrie Asmara, R., Ridwan, M. and Budiprasetyo, G., 2021. Haar cascade and convolutional neural network face detection in client-side for cloud computing face recognition [Online]. *2021 international conference on electrical and information technology (ieit)*. pp.1–5. Available from: https://doi.org/10.1109/IEIT53149.2021.9587388.

Barla, N., 2022. *The complete guide to object tracking [+v7 tutorial]* [Online]. Available from: https://www.v7labs.com/blog/object-tracking-guide.

Bochinski, E., Eiselein, V. and Sikora, T., 2017. High-speed tracking-by-detection without using image information [Online]. *2017 14th ieee international conference on advanced video and signal based surveillance (avss)*. pp.1–6. Available from: https://doi.org/10.1109/AVSS.2017.8078516.

Braulio Rios, Marcos Toscano, A.D., 2020. *Face mask detection in street camera video streams using ai: behind the curtain* [Online]. Available from: https://tryolabs.com/blog/2020/07/09/face-mask-detection-in-street-camera-video-streams-using-ai-behind-the-curtain.

Camlytics, 2022. *People counting solution* [Online]. Available from: https://camlytics.com/solutions/people-counting.

Canu, S., 2021. *How to count people from cctv camerass* [Online]. Available from: https://pysource.com/2021/12/07/how-to-count-people-from-cctv-cameras-with-opencv-and-deep-learning/.

Cloudmatrix, 2022. People counting system. https://www.cloudmatrix.com.tw/en/solution-people-counting-system/.

Code, P. with, 2021. Face detection. https://paperswithcode.com/task/face-detection.

Deb, C., 2020. Face mask detection. https://github.com/chandrikadeb7/Face-Mask-Detection.

Ding, Y., Li, Z. and Yastremsky, D., 2021. Real-time face mask detection in video data [Online]. [Online]. Available from: https://doi.org/10.48550/ARXIV.2105.01816.

Employee, n.d. *Access ip camera in python opencv* [Online]. Available from: https://stackoverflow.com/questions/49978705/access-ip-camera-in-python-opencv.

Eocortex, 2022. Face mask detector. https://eocortex.com/products/video-management-software-vms/face-mask-detector-1.

FootfallCam, 2017. *223 evolution of people counters* [Online]. Available from: https://www.footfallcam.com/blog/2017/12/223-evolution-of-people-counters/.

Fransiska, 2019. *Inception, resnet, mobilenet* [Online]. Available from: https://medium.com/@fransiska26/the-differences-between-inception-resnet-and-mobilenet-e97736a709b0.

Goyal, H., S.K.S.C.e.a., 2022. A real time face mask detection system using convolutional neural network [Online]. [Online]. Available from: https://doi.org/10.1007/s11042-022-12166-x.

Hukkelås, H., 2021. Face-detection 0.2.2. https://pypi.org/project/face-detection/.

Karchaudhuri, P., 2020. *Detecting masks in dense crowds in real-time with ai and computer vision* [Online]. Available from: https://towardsdatascience.com/detecting-masks-in-dense-crowds-in-real-time-with-ai-and-computer-vision-9e819eb90

Khandelwal, R., 2022. *A centroid-based object tracking implementation* [Online]. Available from: https://medium.com/aiguys/a-centroid-based-object-tracking-implementation-455021c2c997.

Khandelwal, Y., 2021. *Crowd counting using deep learning* [Online]. Available from: https://www.analyticsvidhya.com/blog/2021/06/crowd-counting-using-deep-learning/.

Kumar, A., 2021. *Face mask detection using ssd* [Online]. Available from: https://www.kaggle.com/code/aman10kr/face-mask-detection-using-ssd/notebook.

LeewayHertz, 2022. Face mask detection system using artificial intelligence. https://www.leewayhertz.com/face-mask-detection-system/.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.Y. and Berg, A.C., 2016. SSD: Single shot MultiBox detector [Online]. *Computer vision – ECCV 2016*. Springer International Publishing, pp.21–37. Available from: https://doi.org/10.1007/978-3-319-46448-0_2.

LTD, A.T., 2022. *People counting by axiomatic* [Online]. Available from: https://peoplecounting.co.uk/.

Meel, V., 2021. *Object tracking in computer vision (complete guide)* [Online]. Available from: https://viso.ai/deep-learning/object-tracking/.

Meraki, 2022. Face mask detection. https://apps.meraki.io/en-US/apps/381402/face-mask-detection.

Mesnik, B., 2019. *Ip camera systems – the complete reference guide* [Online]. Available from: https://kintronics.com/ip-camera-systems-complete-reference-guide/.

Metro, 2022. *The evolution of people counters: How do they work?* [Online]. Available from: https://metroxp.com/the-evolution-of-people-counters-how-do-they-work/.

Minolta, K., 2022. Face mask detection. https://kmbs.konicaminolta.us/solutions-services/video-security-solutions/face-mask-detection/.

Organization, W.H., 2021. *Coronavirus disease (covid-19) advice for the public: When and how to use masks* [Online]. Available from: https://www.who.int/emergencies/diseases/novel-coronavirus-2019/advice-for-public/when-and-how-to-use-masks.

Padmashini, M., Manjusha, R. and Parameswaran, L., 2018. Vision based algorithm for people counting using deep learning. *International journal of engineering technology*, 7(3.6), pp.74–80.

Patil, V., 2021. *Yolo face* [Online]. Available from: https://pypi.org/project/yoloface/.

Pokhrel, S., 2019. *How to set up real-time face detection on raspberry pi* [Online]. Available from: https://xailient.com/blog/how-to-set-up-real-time-face-detection-on-raspberry-pi/.

Rath, S.R., 2021. *Image classification using tensorflow pretrained models* [Online]. Available from: https://debuggercafe.com/image-classification-using-tensorflow-pretrained-models/.

Raydiant, S. by, 2022. Comply with c-19 regulations: Face mask detection. https://sightcorp.com/face-mask-detection/.

Redmon, J. and Farhadi, A., 2018. Yolov3: An incremental improvement. *arxiv*.

Retail, L., 2022. How does people counting work? https://linkretail.com/how-does-people-counting-work/.

Review, W.P., 2022. *Countries with mask mandates 2022* [Online]. Available from: https://worldpopulationreview.com/country-rankings/countries-with-mask-mandates.

Rosebrock, A., 2018. *Opencv object tracking* [Online]. Available from: https://pyimagesearch.com/2018/07/30/opencv-object-tracking/.

Rupesh, n.d. *Face detection models and their performance comparison* [Online]. Available from: https://rupeshthetech.medium.com/face-detection-models-and-their-performance-comparison-eb8da55f328c.

Singh, A., 2021. *Difference between tensorflow and caffe* [Online]. Available from: https://www.geeksforgeeks.org/difference-between-tensorflow-and-caffe/#:~:text=TensorFlow.

Slashdot, 2022. Best people counting software of 2022. https://slashdot.org/software/people-counting/.

Soviany, P. and Ionescu, R.T., 2018. Optimizing the trade-off between single-stage and two-stage object detectors using image difficulty prediction [Online]. [Online]. Available from: https://doi.org/10.48550/ARXIV.1803.08707.

Srinivas, B., 2020. Face mask detection. https://github.com/balajisrinivas/Face-Mask-Detection.

Techvision, A., 2021. *People counting using computer vision and deep learning* [Online]. Available from: https://www.aividtechvision.com/people-counting-using-computer-vision/.

Unadkat, J., 2021. *Manual testing for beginners* [Online]. Available from: https://www.browserstack.com/guide/manual-testing-tutorial.

Vibhuti, Jindal, N., Singh, H. and Rana, P., 2022. Face mask detection in covid-19: a strategic review. *Multimedia tools and applications* [Online]. Available from: https://doi.org/10.1007/s11042-022-12999-6.

Watch, B., 2019. *Types of cctv cameras – the complete guide* [Online]. Available from: https://www.businesswatchgroup.co.uk/types-of-cctv-cameras-the-complete-guide/.

Xu, T., Feng, Z.H., Wu, X.J. and Kittler, J., 2019. Learning adaptive discriminative correlation filters via temporal consistency preserving spatial feature selection for robust visual object tracking. *IEEE transactions on image processing* [Online], 28(11), pp.5596–5609. Available from: https://doi.org/10.1109/tip.2019.2919201.

Yadav, S., 2020. Deep learning based safe social distancing and face mask detection in public areas for covid19 safety guidelines adherence [Online]. [Online]. Available from: https://doi.org/10.22214/ijraset.2020.30560.

Zhenqiu Xiao, B.Y. and Tjahjadi, D., n.d. [Online]. [Online]. Available from: https://file.techscience.com/files/cmc/2019/v60n3/20190827015721_54612.pdf.

-

Appendix A

User Documentation

Steps

0. Ensure that your machine is using Windows.

1. Ensure that the face detector and mask classifier files are in the same folder as detection_counter.exe

2. Click the detection_counter.exe file to run the programme (the programme may take a few minutes to run).

3. If the programme fails to start, it is (most likely) because no webcam was detected (see exception, and reconnect your webcam).

4. When the programme starts running, a new window will appear that shows the camera feed along with detection data.

5. By default, data is exported every 1 hour (to save on memory) along with visualisations. The relevant files will appear with their date-time in the file name in the same folder as the programme.

6. Press "q" to end the programme (recent data is saved automatically when the programme is stopped).

Appendix B

System Parameters

Parameter	Value	Notes
Video Frame Width	900	Height is set dynamically to preserve aspect ratio
Boundary Line Placement	0.35 * Frame Width	Position of vertical line
Data Export Interval	1 hour	How often the data is exported and memory is cleared
Video Display Font Size	min(width, height) / (25/0.030)	Suits a variety of aspect ratios
Classifier Uncertain Interval	0.2	

Table B.1: Chosen values for all system parameters

Appendix C

Code

Note: The testing code is included in the submitted files (to not reveal the identity of participants).

NOTE For this to typeset correctly, ensure you use the pdflatex command in preference to the latex command. If you do not have the pdflatex command, you will need to remove the landscape and multicols tags and just make do with single column listing output

C.1 Main file

This file is the entrypoint of the program.

```
from detector_counter.detector_counter import run

if __name__ == "__main__":
    run()
```

C.2 Detector Counter

This file contains the main execution loop of the system.

```
import numpy as np
import pandas as pd
import seaborn as sns
import imutils
from imutils.video import VideoStream
import cv2
import time
import matplotlib.pyplot as plt
from datetime import datetime
import tensorflow as tf
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.models import load_model
from detector_counter.tracker_counter import TrackerCounter
from detector_counter.detect_and_classify import detect_and_classify


def run():
  classifier = tf.keras.models.load_model('mask_classifier_mobilenet.h5')

  prototxtPath = r"face_detector_ssd_caffee/deploy.prototxt" # configurations file
  weightsPath = r"face_detector_ssd_caffee/res10_300x300_ssd_iter_140000.caffemodel" # weights file
  detector = cv2.dnn.readNet(prototxtPath, weightsPath) # load saved model

  vs = VideoStream(src=0).start()

  tracker = TrackerCounter()

  # Initialise total tracked faces and fps to 0
  idd = 0
  fps_start_time = 0
  fps = 0

  # Parameters
  W = 900
  left_boundary_line = int(0.35*W)

  # Store starting time for periodic data extraction
  start_time = datetime.now()
  hrs = 1 # every how many hours should the data be exported

  # loop over the frames from the video stream
  while True:
```

```
frame = vs.read()

'''VISUALS'''

# Resize the frame
# NB: cv2.resize lets you choose height and width, imutils.resize only lets you choose width but preserves ratio
# frame = cv2.resize(frame, dsize=(x, y))
frame = imutils.resize(frame, width=W)
H, W = frame.shape[:2] # used in detect_and_classify also

# Calculate ideal font scale
scale = 0.030                    # this value can be from 0 to 1 (0,1] to change the size of the text relative to the image
ideal_font_size = min(W,H)/(25/scale)

# Define box locations
yy = int(ideal_font_size*35)
box1start = np.array((0, 0))
box1end = np.array((int(0.40*W), yy*2))
box2start = np.array((int(0.40*W), 0))
box2end = np.array((W, int(yy*4.5)))

cv2.rectangle(frame, pt1=box1start, pt2=box1end, color=(255, 255, 255), thickness=-1)
cv2.rectangle(frame, pt1=box2start, pt2=box2end, color=(255, 255, 255), thickness=-1)

'''DETECT & CLASSIFY'''
# Detect and Classify for each frame
locs, preds= detect_and_classify(frame, detector, classifier, W=W, H=H)

'''VISUALS'''

label_probs = [] # This is for tracker_counter to average over multiple classifications
# Live Counter
num_of_masked = 0
num_of_unmasked = 0
num_of_uncertain = 0

# Define classification uncertainty interval
uncertain_interval = 0.2 # 0.5 means 50+% probability of a class for classification.

# loop over face locations and mask predictions
for box, pred in zip(locs, preds):

    # unpack the bounding box and predictions
    startX, startY, endX, endY = box
    mask, no_mask = pred
```

```
    # 1. Determine the class label 2. Add colour (BGR)
    if mask >= 0.5 + uncertain_interval:
        label = 'Mask'
        colour = (0, 255, 0)
        num_of_masked +=1
        # for tracker_counter to average over
        label_probs.append(mask)

    elif no_mask >= 0.5 + uncertain_interval:
        label = 'No Mask'
        colour = (0, 0, 255)
        num_of_unmasked +=1
        # for tracker_counter to average over
        label_probs.append(-no_mask)

    elif (mask >= 0.5) and (mask <= 0.5 + uncertain_interval):
        label = 'Uncertain'
        colour = (0, 255, 255)
        num_of_uncertain +=1
        # for tracker_counter to average over
        label_probs.append(mask)

    elif (no_mask >= 0.5) and (no_mask <= 0.5 + uncertain_interval):
        label = 'Uncertain'
        colour = (0, 255, 255)
        num_of_uncertain +=1
        # for tracker_counter to average over
        label_probs.append(-no_mask)

    # probability and text to display
    probability = max(mask, no_mask) * 100
    label_text = f'{label}: {probability:.1f}%'

    # 1. Display label
    cv2.putText(img=frame, text=label_text, org=(startX, startY - 15), fontFace=cv2.FONT_HERSHEY_SIMPLEX, fontScale=0.7,
        color=colour, thickness=2)
    # 2. Display bounding box
    cv2.rectangle(img=frame, pt1=(startX, startY), pt2=(endX, endY), color=colour, thickness=2)

'''TRACK & COUNT'''

# objects_info = [object1_info, object2_info, ...] where object_info = [Xstart, Ystart, Xend, Yend, id_of_object]
objects_info = tracker.update(frame, locs, label_probs, W, H, left_boundary_line, uncertain_interval=uncertain_interval,
    dist_same_obj=(W + H / 14))

'''VISUALS'''
```

```
# for all objects
for object_info in objects_info:
    Xstart, Ystart, Xend, Yend, idd = object_info

    # ID of face
    cv2.putText(img=frame, text=f'Face␣{idd}', org=(Xstart, Ystart-40), fontScale=1.4, fontFace=cv2.FONT_HERSHEY_PLAIN,
        color=(155, 149, 24), thickness=2)

# Calculate fps
fps_end_time = time.time()
time_diff = fps_end_time - fps_start_time
fps = int(1/time_diff)
fps = f'FPS:␣{fps}'
fps_start_time = fps_end_time

# Calculate current time and export data
time_difference = (datetime.now() - start_time).seconds/3600

if  time_difference >= hrs:
  _export_data(tracker)

  # reset time
  start_time = datetime.now()

# VISUALS: Display FPS and Current Time
cv2.putText(img=frame, text=_get_current_time_str(), org=(box1start[0] + box1start[0]//14, yy), fontFace=cv2.FONT_HERSHEY_SIMPLEX,
    fontScale=ideal_font_size, color=(0, 0, 0), thickness=2)
cv2.putText(img=frame, text=fps, org=(box1start[0] + box1start[0]//14, int(yy*1.8)), fontFace=cv2.FONT_HERSHEY_SIMPLEX,
    fontScale=ideal_font_size, color=(0, 0, 0), thickness=2)

# VISUALS: Display boundary line
cv2.line(img=frame, pt1=(left_boundary_line, 0), pt2=(left_boundary_line, H), color=(45, 174, 102), thickness=5)

# VISUALS: Display live people counter
cv2.putText(img=frame, text=f'People␣Count:␣{tracker.people_count}', org=(box2start[0] + box2start[0]//15, yy),
    fontFace=cv2.FONT_HERSHEY_SIMPLEX, fontScale=ideal_font_size, color=(0, 0, 0), thickness=2)
percent_masked = 0
percent_unmasked = 0
percent_uncertain = 0
if tracker.people_count != 0:
    percent_masked = np.round(tracker.mask_count / tracker.people_count * 100, 1)
    percent_unmasked = np.round(tracker.nomask_count / tracker.people_count * 100, 1)
    percent_uncertain = np.round(tracker.uncertain_count / tracker.people_count * 100, 1)
cv2.putText(img=frame, text=f'Masked:␣␣␣␣␣␣␣{tracker.mask_count}␣({percent_masked}%)', org=(box2start[0] + box2start[0]//15,
    yy*2), fontFace=cv2.FONT_HERSHEY_SIMPLEX, fontScale=ideal_font_size, color=(0, 155, 0), thickness=2)
cv2.putText(img=frame, text=f'Unmasked:␣␣␣␣{tracker.nomask_count}␣({percent_unmasked}%)', org=(box2start[0] + box2start[0]//15,
    int(yy*3)), fontFace=cv2.FONT_HERSHEY_SIMPLEX, fontScale=ideal_font_size, color=(0, 0, 155), thickness=2)
```

```
        cv2.putText(img=frame, text=f'Uncertain:␣␣␣␣␣{tracker.uncertain_count}␣({percent_uncertain}%)', org=(box2start[0] +
            box2start[0]//15, int(yy*4)), fontFace=cv2.FONT_HERSHEY_SIMPLEX, fontScale=ideal_font_size, color=(0, 155, 155), thickness=2)

        '''END␣STREAM'''

        # Show the output frame in real-time
        cv2.imshow("Frame", frame)

        # Terminate if 'q' is pressed. waitKey(0): keeps image still until a key is pressed. waitKey(x) it will wait x miliseconds each
            frame
        key = cv2.waitKey(1) & 0xFF
        if key == ord("q"):
            break

    '''EXPORT␣RESULTS'''
    _export_data(tracker)


    # Cleanup
    vs.stop()
    cv2.destroyAllWindows()


def _export_data(tracker):
    current_time_export = _get_current_time_str().replace(':', '.')

    #Export csv file and summarise results
    detection_data_exp = pd.DataFrame(data=tracker.detection_data, columns=['Label', 'Datetime'])
    detection_data_exp.to_csv(f'{current_time_export}_detection_data.csv', index=False)

    # Unburden memory
    tracker.detection_data = []

    # Summarise and visualise results
    num_of_people = detection_data_exp.shape[0]
    if num_of_people > 0:
        mask = detection_data_exp[detection_data_exp.Label == 'mask'].shape[0]
        no_mask = detection_data_exp[detection_data_exp.Label == 'no_mask'].shape[0]
        uncertain = detection_data_exp[detection_data_exp.Label == 'uncertain'].shape[0]

        print(f'{num_of_people}␣people.')
        print(f'{mask}␣mask.')
        print(f'{no_mask}␣no_mask.')
        print(f'{uncertain}␣uncertain.')

        # PIE CHART
        dpi=110
```

```
        fig = plt.figure(figsize=(8, 6), dpi=dpi)

        if uncertain != 0:
            plt.pie(x=[mask, no_mask, uncertain], labels=['Mask', 'No Mask', 'Uncertain'], colors=['green', 'red', 'yellow'],
                startangle=90, autopct='%1.1f%%', textprops={'fontsize': 14})
        if uncertain == 0:
            plt.pie(x=[mask, no_mask], labels=['Mask', 'No Mask'], colors=['green', 'red'], startangle=90, autopct='%1.1f%%',
                textprops={'fontsize': 14})

        plt.title(f'Total People: {num_of_people}', fontweight='bold', fontsize=15)
        plt.legend()
        plt.show()

        fig.savefig(f'{current_time_export}_face_covering_pie_chart.png', dpi = dpi)

    else:
        print('No people detected.')
        pass


def _get_current_time_str():
    return str(datetime.now())[:-7]
```

C.3 Detect & Classify

This file contains the face-detection and mask-classification code.

```
import cv2
import numpy as np
import tensorflow as tf


def detect_and_classify(frame, detector, classifier, W, H, conf = 0.3):
    '''
    Input: video frame, face detection model, face mask classification model
    Output: detector bounding boxes, classifier predictions, bounding boxes for tracker

    detect faces in frame and perform mask classification
    turn frame into blob (essentially preprocessing: 1. mean subtraction, 2. scaling, 3. optionally channel swapping)
    '''

    RGB = (104.0, 177.0, 123.0) # Source and intuition: "Deep Learning and Mean Subtraction":
        https://pyimagesearch.com/2017/11/06/deep-learning-opencvs-blobfromimage-works/
    pixels = 224
    blob = cv2.dnn.blobFromImage(frame, scalefactor=1.0, size=(pixels, pixels), mean=RGB)

    ''' DETECTION '''
    # Detect Faces
    detector.setInput(blob)
    detections = detector.forward()
    num_of_detections = detections.shape[2]

    # initialise list of faces, their locations, list of predictions for our mask classifier
    faces = []
    locs = []
    preds = []

    # loop over all face detections
    for i in range(num_of_detections):

        # Live Counter to display on video feed
        num_of_masked = 0
        num_of_unmasked = 0
        num_of_uncertain = 0

        # extract the confidence (probability) in the detection
        confidence = detections[0, 0, i, 2]

        # filter out weak detections by ensuring the confidence is  greater than the minimum confidence
        if confidence > conf:
```

```
        # compute the (x, y)-coordinates of the bounding box for the object
        box = detections[0, 0, i, 3:7] * np.array([W, H, W, H])
        (startX, startY, endX, endY) = box.astype("int")

        # ensure the bounding boxes fall within the dimensions of the frame
        (startX, startY) = (max(0, startX), max(0, startY))
        (endX, endY) = (min(W - 1, endX), min(H - 1, endY))

        # Extract Face
        face = frame[startY:endY, startX:endX]                          # 1. extract the face region of interest (ROI)
        face = cv2.cvtColor(face, cv2.COLOR_BGR2RGB)                    # 2. convert it from BGR to RGB channel ordering
        face = cv2.resize(face, (224, 224))                             # 3. resize it to 224x224
        face = tf.keras.preprocessing.image.img_to_array(face)          # 4. preprocess it for the mask classifier
        face = tf.keras.applications.mobilenet_v2.preprocess_input(face)

        # append face and bounding box to lists
        faces.append(face)
        locs.append(np.array([startX, startY, endX, endY]))

''' CLASSIFICATION '''
# only make a predictions if at least one face was detected
if len(faces) > 0:
    # for faster inference make batch predictions on *all* faces at the same time rather than one-by-one predictions in the above
        'for' loop
    faces = np.array(faces, dtype="float32")
    preds = classifier.predict(faces, batch_size=32, verbose=0)

return locs, preds
```

C.4 Tracker Counter

This file contains the implementation of the Tracker Counter component.

```
import math
from random import randrange
import numpy as np
from datetime import datetime
import cv2
from PIL import Image
import matplotlib.pyplot as plt


class TrackerCounter:
  people_count = 0
  mask_count = 0
  nomask_count = 0
  uncertain_count = 0

  def __init__(self):
    # Store information for each object in a dictionary
    # values: [cx (int), cy (int), has_been_before_boundary (boolean), has_been_counted_before (boolean), list_of_up_to_10_predictions
        (float [-1, 1])]
    self.center_points = {}

    # each time a new object id detected, the count will increase by one
    self.id_count = 0

    self.detection_data = []

  def update(self, frame, objects_rect, label_probs, W, H, left_boundary_line, uncertain_interval=0.2, dist_same_obj=100):
    '''
    object_rect: List of objects' coordinates = [(xstart1, ystart1, xend1, yend1), (xstart2, ystart2, xend2, yend2), ...]
    label_probs: List of objects' label "probabilities" = [-0.9, 0.7, ....]
    W, H: frame width and height

    dist_same_obj:
    - It is the maximum eucledian distance from the previous object in order to be considered the same object
    - dist_same_obj determines how far the detection has to be from the old one to be considered a new object.
    - If dist_same_obj is too low we might get false positives when an object is moving.
    - It also depends on the amount of pixels so needs different value when resolution changes.
    '''

    # List of info of all objects in the frame (this is what the method returns)
    objects_infos = []
```

```
    # NB: If dist_same_obj is too low then fast moving faces are seen as a new face in each frame and averaging classification is not
        done OR even worse they are not counted at all (because a new face appears after the boundary)
    # NB: If dist_same_obj is too high then if a face disappears at frame 15 and a new face appears at frame 16 (it is only a problem if
        this happens in consecutive frames since memory is not implemented i.e. we got 1 frame memory)
    # then the new face will be seen as the old face and might not be tracked
    # NB: This is very unlikely to be an issue when memory = 1 as it is now so it is a lot safer to have a high dist_same_obj than a low
        one.
    dist_same_obj=(W+H)/10


    counter=0 # This variable represents how many faces of previous frame have been matched to faces of the new frame (e.g. if counter =
        2 it means that 2 faces from the previous frame have been matched to 2 of the new frame). This makes sure two or more faces are
        not labelled the same

      # Loop through all faces and their labels
    for rect, prob in zip(objects_rect, label_probs):
        xstart, ystart, xend, yend = rect
        # Get center point of new object
        cx = (xstart + xend) // 2
        cy = (ystart + yend) // 2


        '''Find␣out␣if␣same␣face␣was␣detected'''
        same_object_detected = False
        # For all objects in previous frame
        for face_id, pt in self.center_points.items():

            # calculate eucledian distance from previously detected face
            dist = math.hypot(cx - pt[0], cy - pt[1])

            # 1st condition: If it is the same object (ditsance < dist_same_obj) update previous objects location to this one's␣(keeps␣
                the␣object␣id).
␣␣␣␣␣␣␣␣␣␣␣␣#␣2nd␣condition:␣But␣create␣a␣NEW␣object␣if␣all␣faces␣from␣previous␣frame␣have␣been␣matched␣up␣already␣(i.e.␣we␣if␣we've run
    out of faces to match).
            # otherwise youll get the bug where when 2 people show up it is Face 1 and Face 1 as if they are the same face. So current #
                of objects has to be <= previous number of objects to match to previous objects which makes sense.
            if dist < dist_same_obj and counter < len(self.center_points):
                self.center_points[face_id][0] = cx      # update center x
                self.center_points[face_id][1] = cy      # update center y

                # Append classification probability to this face's␣classification␣history
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣frames_to_avg_over␣=␣10
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣len(pt[4])␣<␣frames_to_avg_over:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.center_points[face_id][4].append(prob)

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣If␣there␣are␣already␣10␣probabilities␣in␣the␣list␣then␣replace␣one␣at␣random␣(because␣the␣code␣for␣cycling␣through␣
    them␣one␣by␣one␣seems␣too␣complex␣in␣this␣case)
```

```
                    else:
                        self.center_points[face_id][4][randrange(frames_to_avg_over)] = prob

                    # Add object info to list to return
                    objects_infos.append([xstart, ystart, xend, yend, face_id])
                    same_object_detected = True

                    counter+=1

                    break

        '''New face detected'''
        # New object is detected: assign ID to that object
        if same_object_detected == False:
            self.id_count += 1
            self.center_points[self.id_count] = [cx, cy, False, False, [prob]]

            # Add new object to list to return
            objects_infos.append([xstart, ystart, xend, yend, self.id_count])

    # Clean the dictionary by center points to remove IDS not used anymore
    new_center_points = {}
    for object_info in objects_infos:
        _, _, _, _, object_id = object_info
        center = self.center_points[object_id]
        new_center_points[object_id] = center

    # Update dictionary with IDs not used removed
    self.center_points = new_center_points.copy()

    # People counter
    for face_id, pt in self.center_points.items():
        # Note if object has been before the boundary line
        if pt[0] > left_boundary_line:
            self.center_points[face_id][2] = True

        # Note if the object
        # 1. has been detected after the boundary
        # 2. has been detected before the boundary (can remove this condition, it is just to ensure that someone who appears from the
            bottom is not counted) and
        # 3. has never been counted before
        # Then update it to having been counted and increase people count by 1
        # NB: pt[0] = xcenter, pt[1] = ycenter
        if (pt[0] < left_boundary_line) and (pt[2] == True) and pt[3] == False:
            self.center_points[face_id][3] = True
            self.people_count += 1
```

```
            div = 3 # the larger div is the smaller the perpetrator frame will be

            # labeled count
            avg_prob = np.mean(pt[4])

            current_time = datetime.now()

            if avg_prob > 0. + uncertain_interval:
                self.mask_count += 1

                # TODO: Only for testing, remove
                # Save and display image of masked person
                print(f'MASK ({current_time})')
                self.detection_data.append(['mask', current_time])
                # Catch exception where face is moving too fast and towards bottom left and perpetrator[0] or perpetrator[1] (width or
                    height or both) ends up being 0.
                perpetrator=frame[0:H, 0:left_boundary_line + W//5]
                perpetrator = cv2.cvtColor(perpetrator, cv2.COLOR_BGR2RGB)
                perpetrator = Image.fromarray(perpetrator, 'RGB')
                plt.imshow(perpetrator)
                plt.show()

            elif avg_prob < -0. - uncertain_interval:
                self.nomask_count += 1

                # Save and display image of unmasked person
                print(f'NO MASK ({current_time})')
                self.detection_data.append(['no_mask', current_time])
                # Catch exception where face is moving too fast and towards bottom left and perpetrator[0] or perpetrator[1] (width or
                    height or both) ends up being 0.
                perpetrator=frame[0:H, 0:left_boundary_line + W//5]
                perpetrator = cv2.cvtColor(perpetrator, cv2.COLOR_BGR2RGB)
                perpetrator = Image.fromarray(perpetrator, 'RGB')
                plt.imshow(perpetrator)
                plt.show()

            else:
                self.uncertain_count += 1

                # Save and display image of potentially unmasked person
                print(f'UNCERTAIN ({current_time})')
                self.detection_data.append(['uncertain', current_time])
                # Catch exception where face is moving too fast and towards bottom left and perpetrator[1] (the height of the image)
                    ends up being 0.
                perpetrator=frame[0:H, 0:left_boundary_line + W//5]
                perpetrator = cv2.cvtColor(perpetrator, cv2.COLOR_BGR2RGB)
                perpetrator = Image.fromarray(perpetrator, 'RGB')
```

```
                plt.imshow(perpetrator)
                plt.show()

    return objects_infos
```