Log Ruben Stoop

In this document you will find the log of Ruben Stoop containing all of the work that he has contributed towards the Shootsoft project. In the log multiple documents will be referenced. These documents can be found at the following git repository: https://github.com/neburpoots/jupyterfileshootsoft

Week 1

On the 6th of February the project started with a consultation meeting with Annemarie and Micha giving us an introduction to the assignment. The first week was mostly spent on making the action plan and thinking of possible solutions for the assignment. I tried to start with using a couple of computer vision techniques to see what the possible results of these can be. One technique I for instance tried was trying to use feature matching for detecting bullets. I first of all manipulated the image to make the bullets stand out more. To make the bullets stand out more I used the code in the “shooftsofttreshold.ipynb” file. This resulted in an image which looked like Figure 1.

Figure 1. Shootsoft card after image manipulation with “shooftsofttreshold.ipynb”

Using this as a base I tried a multitude of feature matching techniques. The example code of this can be found in “shootsoftfeaturematching.ipynb”. The results of this can be found in the following images. At the top right of each of these images you will find the bullet image template that I used to find the other bullets in the image. The first technique I used was the “cv2.BFMatcher()”. As you can see by looking at Figure 2. It does find some resemblance but not much.

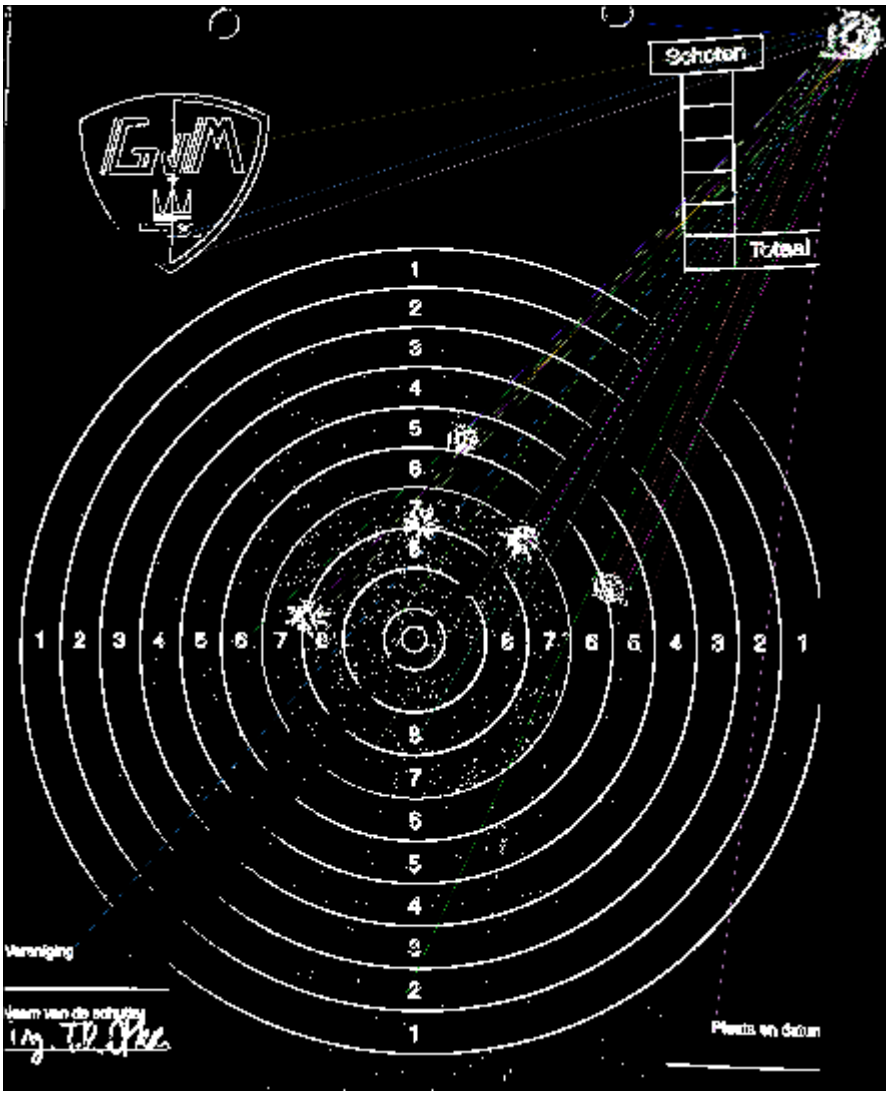

Figure 2. Matches using “cv2.BFMatcher()”

After this I used a different technique by making use of a SIFT with the “cv2.xfeatures2d.SIFT_create()” method. The results are shown in Figure 3. As you can see the results have improved somewhat. But it now also detects the letters as a bullet.

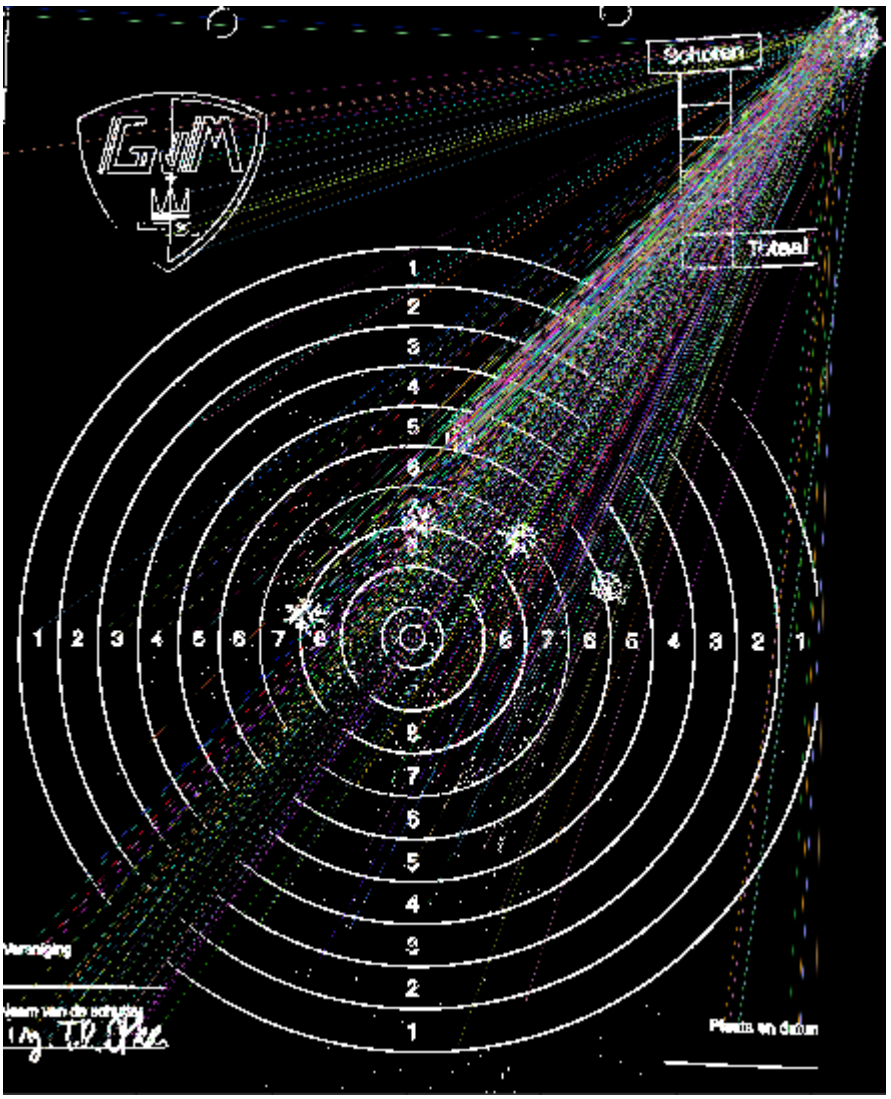

Figure 3. Results of feature matching use “cv2.xfeatures2d.SIFT_create()”

The final technique I used for feature matching was the “cv2.FlannBasedMatcher()”. The results are shown in Figure 4. As you can see the results are decently accurate. You can actually see that it finds the exact bullet of the original image. I decided that this way of detecting bullets deemed to unreliable.

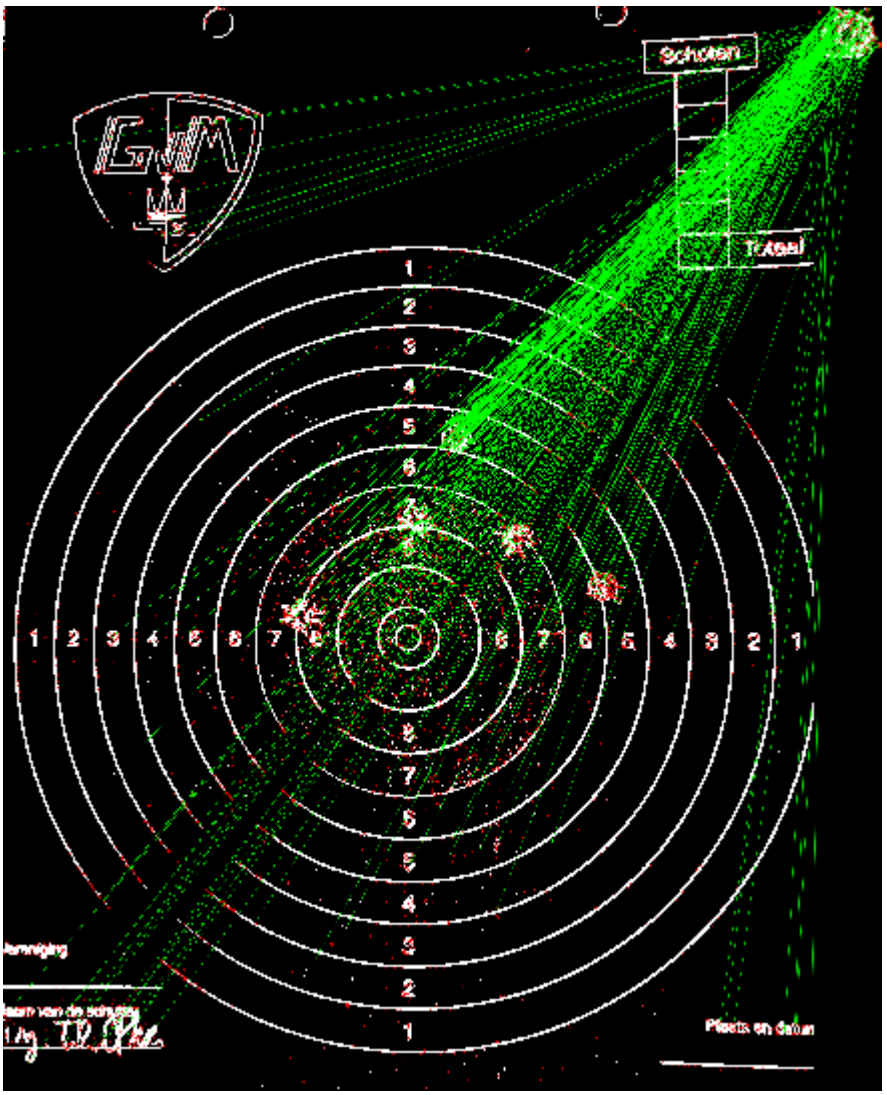

Figure 4. Feature matching using “cv2.FlannBasedMatcher()”

I also tried template matching but this resulted in even worse results. The results of template matching can be found in the “shootsofttemplatedetection.ipynb”. Since the results of the feature matching were mixed I decided that it was probably better to focus on detecting the circles with computer vision and using Artificial intelligence to detect the shots on the shooting card. My plan now was to find a way to detect all circles with computer vision.

Week 2

The second week of the project was spent trying a lot of different computer vision methods to try and detect the circles. I started out with testing the cv2.houghcircles method which resulted in inaccurate results. Some of the examples of this can be found in the “shooftsoftgenerater.ipynb” file. I've tried a multitude of ways by manipulating the image to make the circles more clear. Some of the methods I tried were: Erosion, Dilation, CLAHE, Threshold, AdaptiveTreshold, Blurring. An example code snippet can be found below in Figure 5.

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

kernel = np.ones((5, 5), np.uint8)

img2 = cv2.imread('shootanalysisdataset/schotanalyse/20200923_130907.jpg')
resized_img = cv2.resize(img2, (round((img2.shape[1] / 4)), round((img2.shape[0] / 4))))

kernel = np.ones((5, 5), np.uint8)
img = cv2.erode(resized_img, kernel, iterations=1)

greyresized = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
sep_blur = cv2.blur(greyresized,(8,8))

sep_thresh = cv2.adaptiveThreshold(sep_blur,255,
                                   cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                   cv2.THRESH_BINARY,11,3)
img_erosionv2 = cv2.erode(sep_thresh, kernel, iterations=1)

minDist = 0.0001
param1 = 850 #500
param2 = 300 #200 #smaller value-> more false circles
minRadius = 5
maxRadius = 1200 #10

# docstring of HoughCircles: HoughCircles(image, method, dp, minDist[, circles[, param1[, param2[,
circles = cv2.HoughCircles(img_erosionv2,
                           cv2.HOUGH_GRADIENT,
                           1, minDist,
                           param1=param1,
                           param2=param2,
                           minRadius=minRadius,
                           maxRadius=maxRadius)


if circles is not None:
    circles = np.uint16(np.around(circles))
    for i in circles[0,:]:
        cv2.circle(img2, (i[0], i[1]), i[2], (0, 255, 0), 2)

# Show result for testing:
plt.imshow(img2)
```

Figure 5. Code snippet circle detections with cv2.HoughCircles()

This snippet of code resulted in Figure 6. As you can see this is not really the result you would want. It detects some of the circles of the shooting card but not all. It also detects some of the circles multiple times. This test was also done with a nearly perfect image. The results of other test images were even worse. Further tweaking the parameters of the functions would sometimes yield better results for specific images. But no general solution was found.

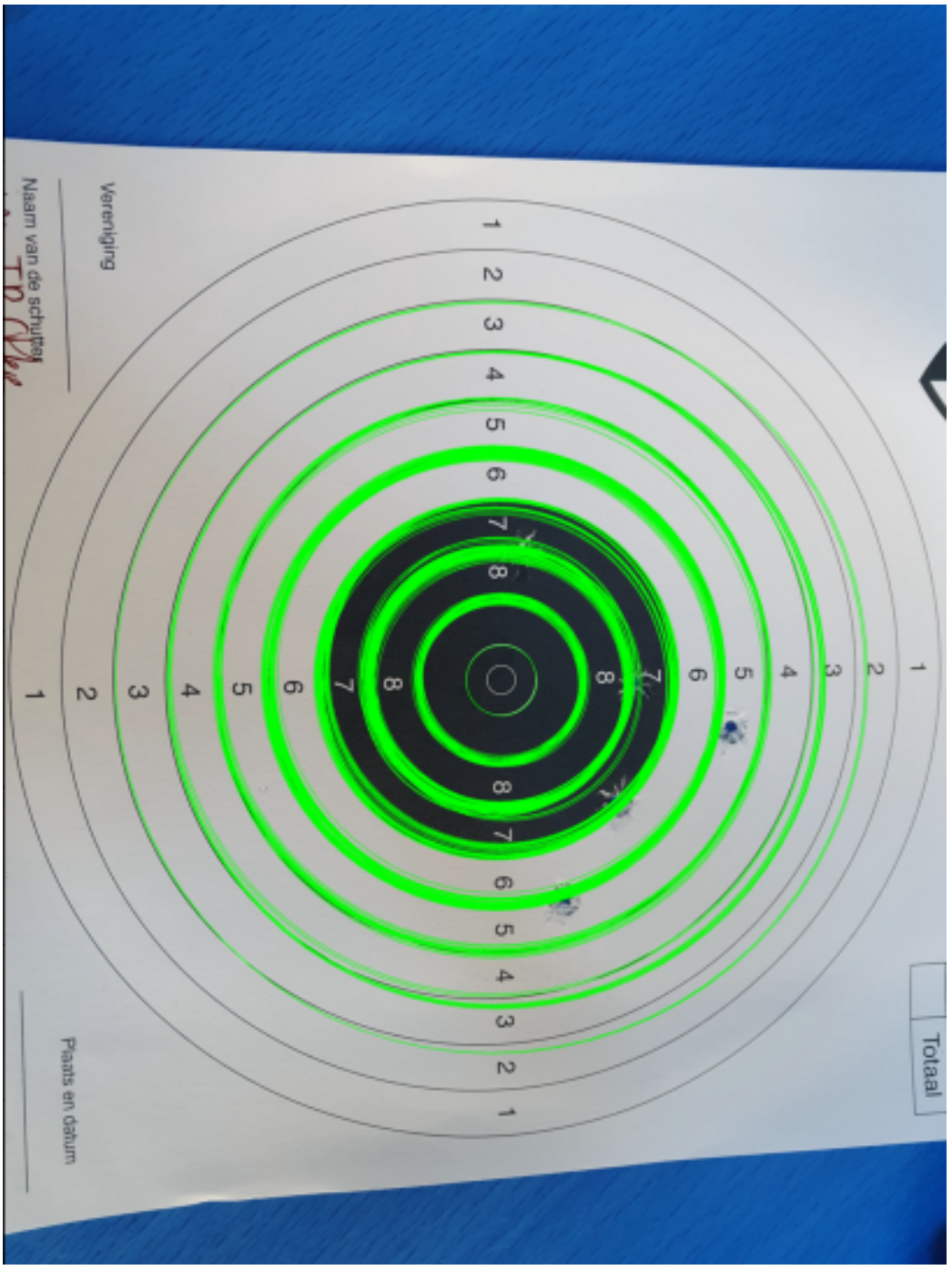

Figure 6. Result of HoughCircles method to detect circles.

This was not the only code snippet I tried for week 2. The “shooftsoftgenerator.ipynb” is full of code snippets with different techniques used. All with mixed results.

Week 3

In the third week I abandoned the strategy of using the methods cv2.houghcircles and opted to see if I could use the black contour to calculate all the circles. To find the black circles I made use of multiple techniques like blurring and the watershed algorithm. A good example of this can be found in the “shootsoftwatershedalgoblackcontour.pynb”. Let's go through some of the steps.

I started with blurring the image and adding a threshold (Figure 7).

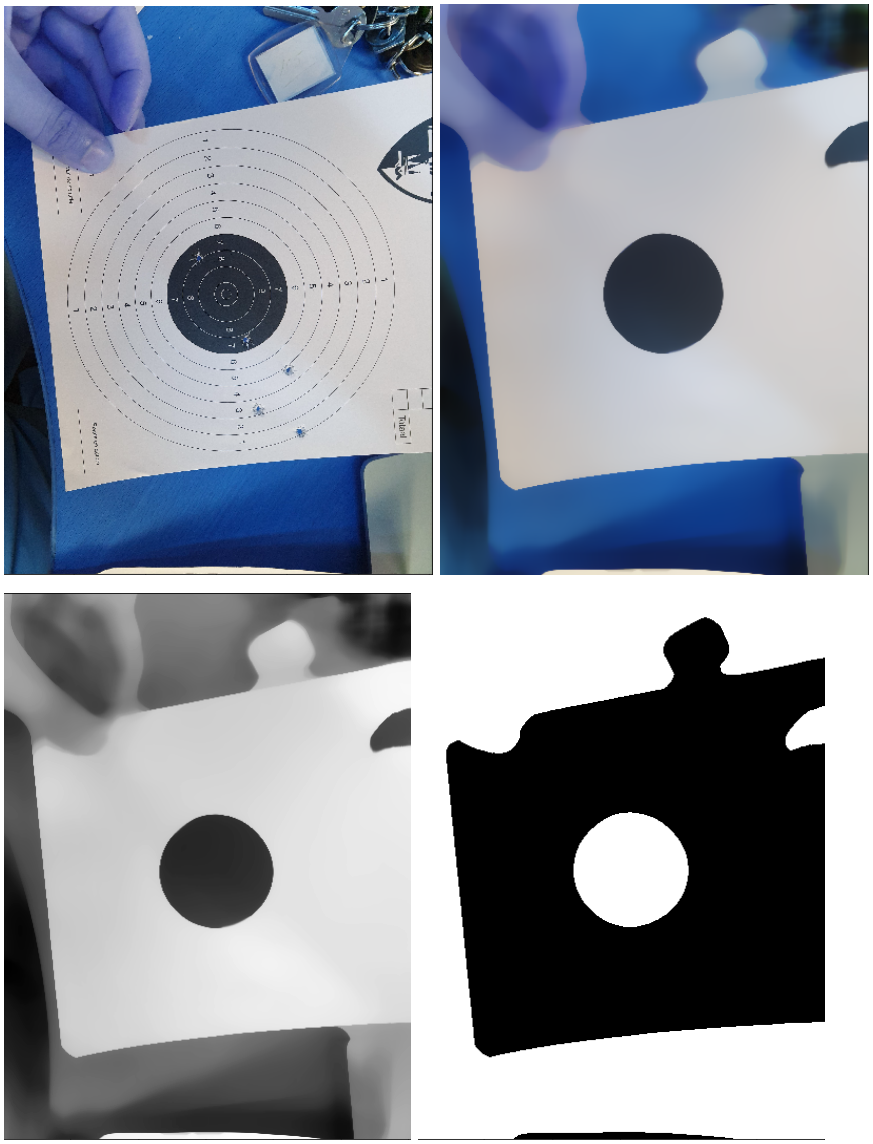

Figure 7. Image manipulation for black contour detection

I then used Figure 8 to detect all contours in the image. The top snippet gives me back an array with found contours and I then do some additional filtering only to use the most complicated contour which is usually the circle.

```
contours,hierarchy = cv2.findContours(sep_thresh.copy(),
                                      cv2.RETR_CCOMP,
                                      cv2.CHAIN_APPROX_SIMPLE)
```

```
contour_list = []
for contour in contours:
    approx = cv2.approxPolyDP(contour,0.01*cv2.arcLength(contour,True),True)
    area = cv2.contourArea(contour)
    if ((len(approx) > 15) & (area > 30) ):
        contour_list.append(contour)


cv2.drawContours(sep_coins, contour_list,  -1, (255,0,0), 2)
plt.imshow(sep_coins)

def scale_contour(cnt, scale):
    M = cv2.moments(cnt)
    cx = int(M['m10']/M['m00'])
    cy = int(M['m01']/M['m00'])

    cnt_norm = cnt - [cx, cy]
    cnt_scaled = cnt_norm * scale
    cnt_scaled = cnt_scaled + [cx, cy]
    cnt_scaled = cnt_scaled.astype(np.int32)

    return cnt_scaled

double_contour = scale_contour(contour_list[0], 3.5)


plt.imshow(sep_coins)
```

Figure 8. Code snippet used to find the black contour in an image.

The result is shown in Figure 9. The initial result looks great and it appears to be working fine in this image. I could find the black circle most of the time but what seems to be a recurring theme with computer vision is that the inconsistencies were just too great. If the picture was taken at an angle for instance calculating the circles was impossible due to the FOV and distortion. If the paper was wrinkled the circles would also be off by a lot.

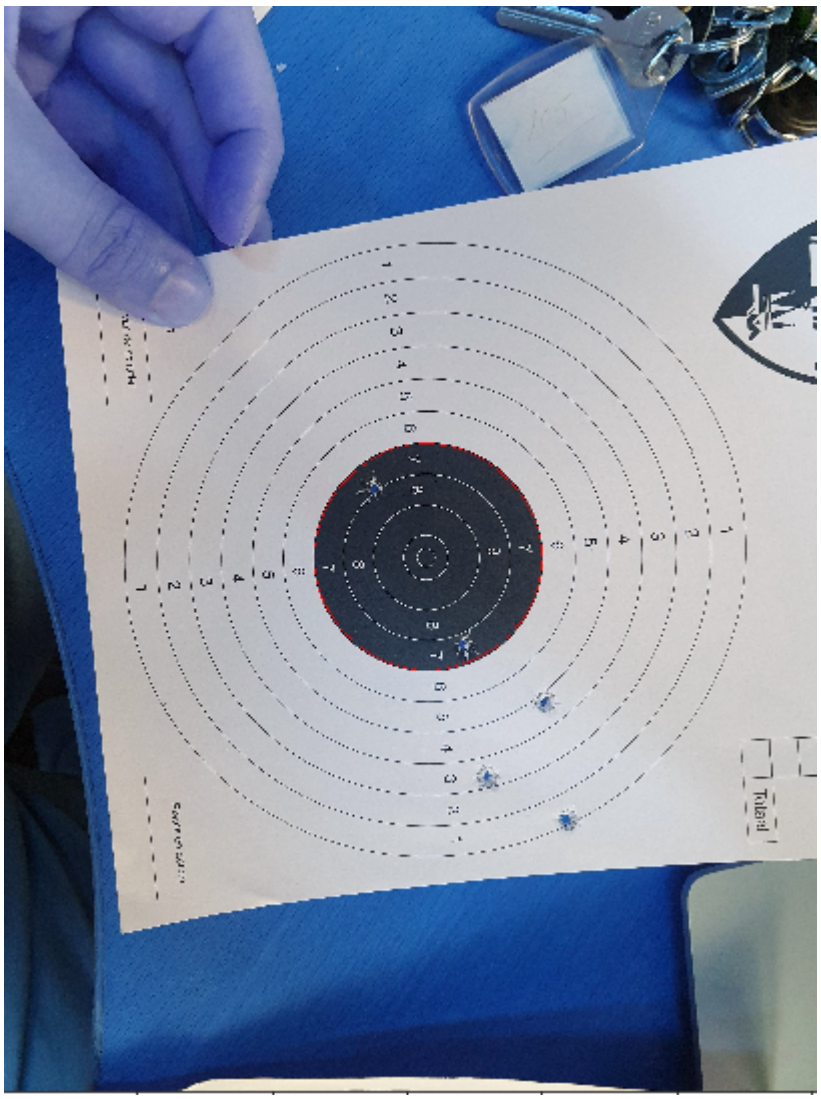

Figure 9. Result of the algorithm to find the black contour.

Week 4

In week 4 I again abandoned the technique of using the black contour to detect all the circles. I did however find the “cv2.findcontours()” method quite intriguing and decided to use it in combination with some parts of the watershed algorithm to detect all circles. The way the “cv2.findcontours()” works is that it finds all contours in an image. However this also detects all contours in the background which are not necessarily needed for detecting circles. Figure 10 is an example of this. All of the examples used in week 4 can be found in the “shootsoftFINAL.ipynb” file.

Figure 10. Contour detection with a lot of extra contours.

To further decrease the amount of contours detected I decided to use the black contour detection from week 3 as a way to crop the image. The example of this is the

“shootsoftCROPPING-Copy1.ipynb”. This finds black contour and then does the dimensions of contour times 3.5. The result of this is shown in Figure 11.

Figure 11. The proposed cropping of an example image.

Now that the image is cropped I used a particular set of image manipulations to make circles as clear as possible while still reducing noise. The final image that the “cv2.findContours()” receives looks like Figure 12. As you can see the lines of the circle appear to be very clear.

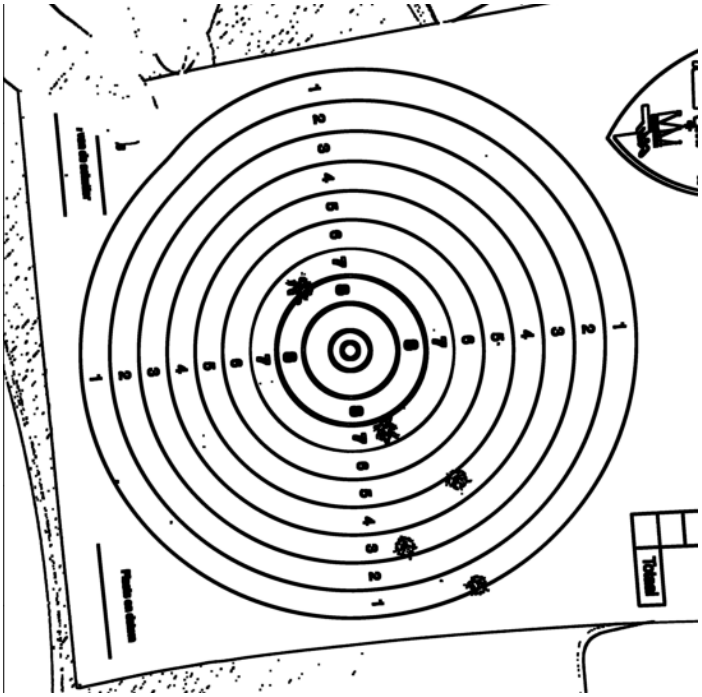

Figure 12. Image after multiple manipulations.

The “cv2.findcontours()” does still seem to find a lot of contours that are not the circles. To try and overcome this problem. I’ve used a multitude of methods to try and remove all contours. The first method I used is to get rid of all contours that were not complicated and big enough to be circles. This resulted in the following snippet of code (Figure 13).

```
for x, contour in enumerate(contours):
    approx = cv2.approxPolyDP(contour,0.01*cv2.arcLength(contour,True),True)
    area = cv2.contourArea(contour)
    if ((len(approx) > 8) & (area > 30000) ):
        contour_list.append([x, contour])
        area_list = np.vstack([area_list, [x, area]])
```

Figure 13. Removing unnecessary contours that are too small.

After this the result still was not good enough. The next idea was to use the center coordinates of each contour. If the center coordinates were too far from the median then the contour will be removed. Circles of course have center coordinates that are nearly perfectly on top of each other. This does depend slightly on the angle of the photo though.

```
#removes contour from contour list if it is not close enough to the medium this i
for coordinate in center_coordinates:

    if(((x_median * 1.2) < coordinate[0] or coordinate[0] < (x_median * 0.8)) or
       ((y_median * 1.2) < coordinate[1] or coordinate[1] < (y_median * 0.8))):

        array_of_keys = np.array([inner_arr[0] for inner_arr in contour_list])
        indices = np.where(array_of_keys == int(coordinate[2]))

        mask = center_coordinates[:, 2] != coordinate[2]
        center_coordinates = center_coordinates[mask]

        contour_list.pop(indices[0][0])
```

Figure 14. Code to remove all contours that are not perfectly in the center.

This still did not result in only the circles being the contours. So I then tried the idea of checking the distance between every single point in the contour and the center coordinates. The idea is illustrated in Figure 15. Anything that is not a circle will have a lot more varied distances between the closest and furthest point from the middle of the contour.

Figure 15. Illustration of checking the distance to the middle for filtering.

The python code to achieve this result is shown in Figure 16. If the distance of point is more than 30% of the average distance then the contour is removed.

```
while counter < (len(contour_list) - 1):
    checkLength = len(contour_list)
    contour = contour_list[counter]
    M = cv2.moments(contour[1])
    cx = int(M['m10']/M['m00'])
    cy = int(M['m01']/M['m00'])
    average = find_inner_array(average_list, contour[0])
    min_average = average[0] * 0.7
    max_average = average[0] * 1.3
    for x, point in enumerate(contour[1]):
        dif_x = cx - point[0][0]
        dif_y = cy - point[0][1]

        dif_x = abs(dif_x)
        dif_y = abs(dif_y)

        distance = math.sqrt((dif_x + dif_y))
        if min_average > distance or max_average < distance:
            print(counter)
            print(average[0])
            print(distance)

            contour_list.pop(counter)
            break

    if checkLength is len(contour_list):
        counter = counter + 1
```

Figure 16. Python code snippet to remove contours with varied lengths in points

After this entire algorithm Figure 17 shows the final results on a perfect example image. The results are basically perfect if you ignore the small indents because of the bullets.

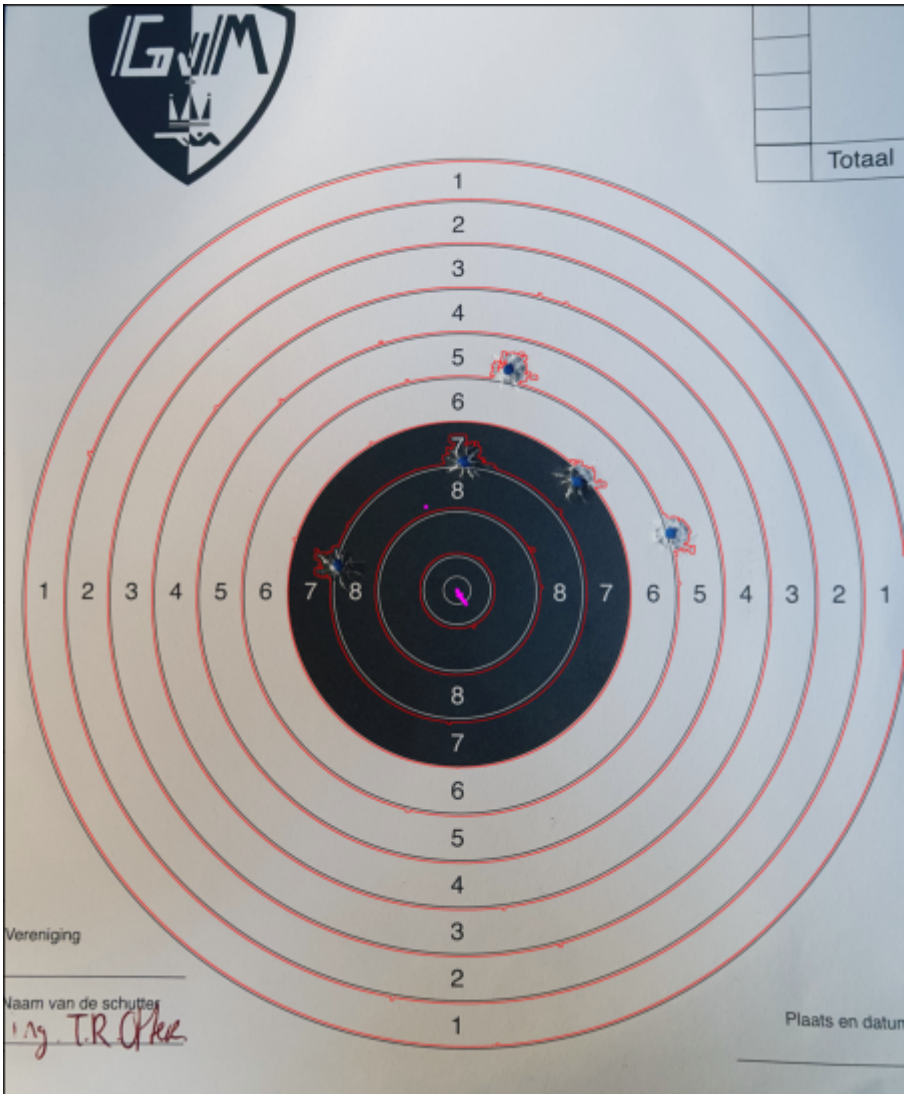

Figure 17. Example of the “ShooftsoftFINAL.ipynb” algorithm result.

The problem with this algorithm though was that it was again too unreliable for less then perfect images. Figure 18 shows a result of a wrinkled shooting card. Some of the contours are now filtered out.

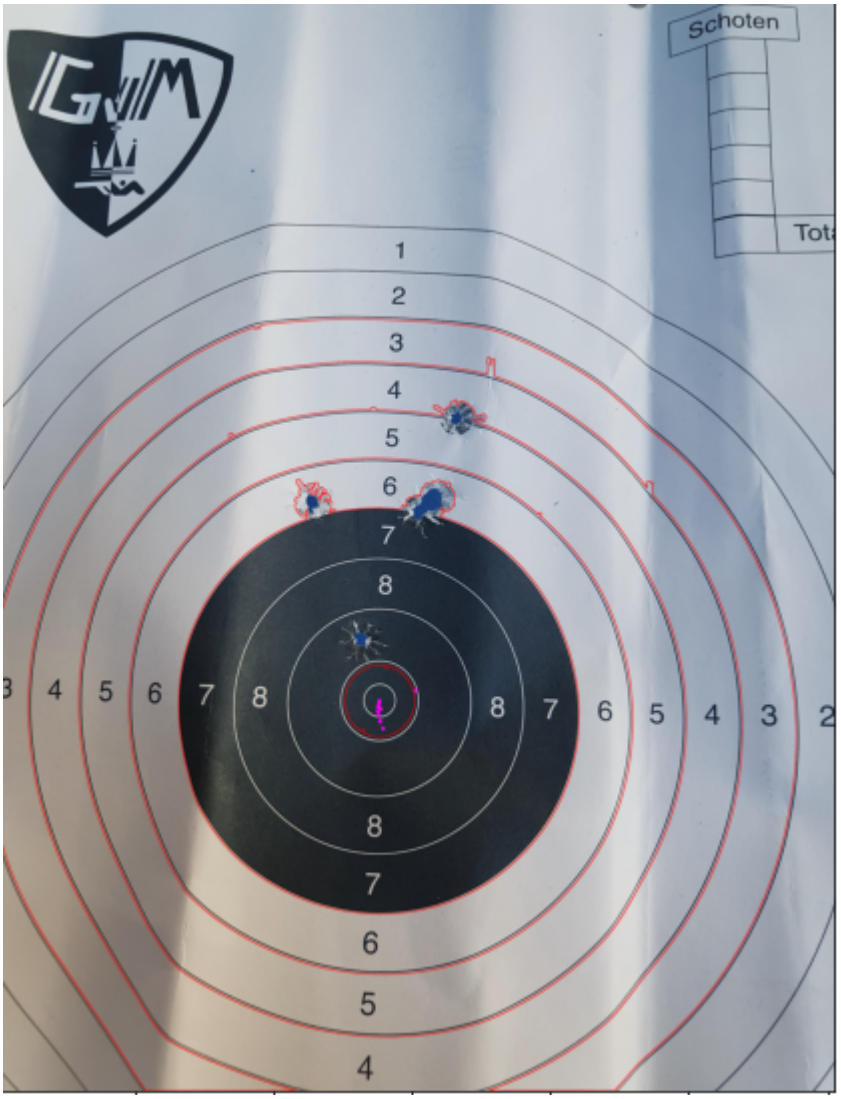

Figure 18. Less than perfect example of the “shootsoftFINAL.ipynb” detection.

As a last resort I tried to generate the circles that were not found in this example by copying the closest circle and then scaling it accordingly. Trying out this method I found that with this exact image. Some of the contours are actually 2 circles because the image is cut off. This made the process incredibly complicated and in the end impossible. The entire algorithm I just went through can be found in “shooftsoftFINAL.ipynb”.

During week 4 Jaimy came with the idea to use Roboflow as a way to annotate our images. Our original dataset that we got from Thijs’s dataset only consisted of sixty images. Jaimy annotated these images in roboflow and trained a model in yolov8. The results of this model were really good and that gave some new hope to the project. introduced us to Roboflow. Roboflow is an online tool for annotating images. Roboflow also allows for exporting a project to Yolov8. This allowed us to detect bullets very accurately.

Week 5

In week 5 we discussed findings in the consultation and with Marya. Marya suggested that instead of detecting bullets we could detect the number of points scored instead. This resulted in us reannotating the images with the score instead of shot. The initial results of this were quite decent with some inaccuracies. This remodeling also allowed us to abandon the idea of using computer vision to detect the circles. The idea quickly arose that we could potentially generate shooting cards. I started with photoshopping all bullets of the current 60 image dataset and this resulted in about 8 different white bullets and about 4 different black bullets. The dataset provided by Thijs consisted actually of only about ⅝ different shooting cards with photos taken from different angles. Since I spent most of my time working with computer vision so far I decided to see if I could generate some extra shooting cards.

In the file “shootsoftGenerationv1.ipynb” you will find the first prototype for the generation of shooting cards. This generator actually takes some inspiration from the circle detection in that it uses the cropping to determine where to generate the shots. You ofcourse don’t want any bullets on the table in the background. The black contour was also important because it has different colored bullets. So to decide which bullets must be white and which must be black. The black contour has to be detected. Figure 19 shows an example image of a shooting card with generated bullets. This example has about 15 shots generated to get the point across.

Figure 19. Example of generated bullets.

The problem with this technique however was that I still had to always find the black contour for this to work properly. So to get a quick and simple solution I decided to use one template image which we knew would work. I then generated around 180 of these images and split the annotations between the three of us. Figure 20 shows an example of an annotated image. Every image has around 5-13 bullets generated on it. I've also tried to use the edge of the black contour as a way of using half black/half white bullets. This sometimes looks very realistic but also sometimes not that much like in this example.

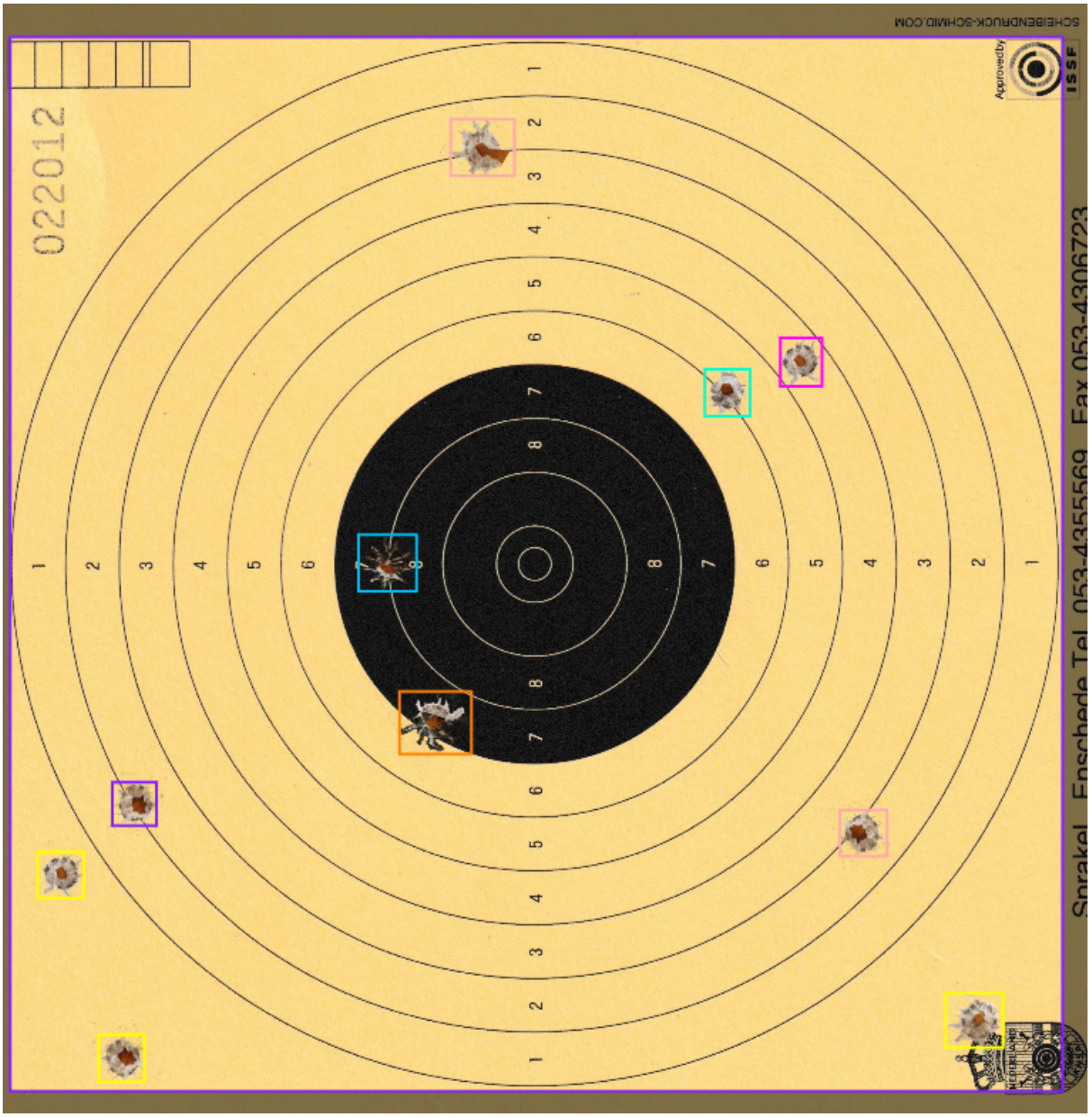

Figure 20. A generated image with annotations

Week 6

In week 6 we had an appointment with Thijs to go to the shooting range to do some field research and collect more examples for our dataset. We got to shoot with some different types of guns but all with .22 millimeter caliber. We also shot some different shooting cards meant for rifles and carbines. We took all of the cards home and we also received some clean cards from Thijs for future uses. Everyone individually took images of their shooting cards and uploaded them to roboflow. Each group member had around 90 images to annotate that we divided equally.

Week 7

At the start of week 7 we had annotated all of our images and the only way to realistically get more data was by generating it. I thought of that plan that could possibly automate the entire steps of the process. It would go something like the following.

1. Take images of empty shooting cards
2. Detect the target which is super reliable with our model by making use of our API.
3. Generate shots within this detected target
4. Send the image with generated shots to our api so that it detects the shots.
5. Receive the coordinates from the shots and generate an annotation file.
6. Send the image with generated shots and annotation to roboflow by making use of the API of roboflow.

Nick introduced me to the API which he had built. This already had most of the dependencies needed and a good menu for usage. I then installed Ultralytics in the api and made two endpoints that we could use to get back results from our model.

One endpoint that takes a file and gives back the coordinates. Figure 21 shows what this would look like in postman. Figure 22 shows the second endpoint which sends a file and receives a file with the detected results. This will be very useful for future testing.

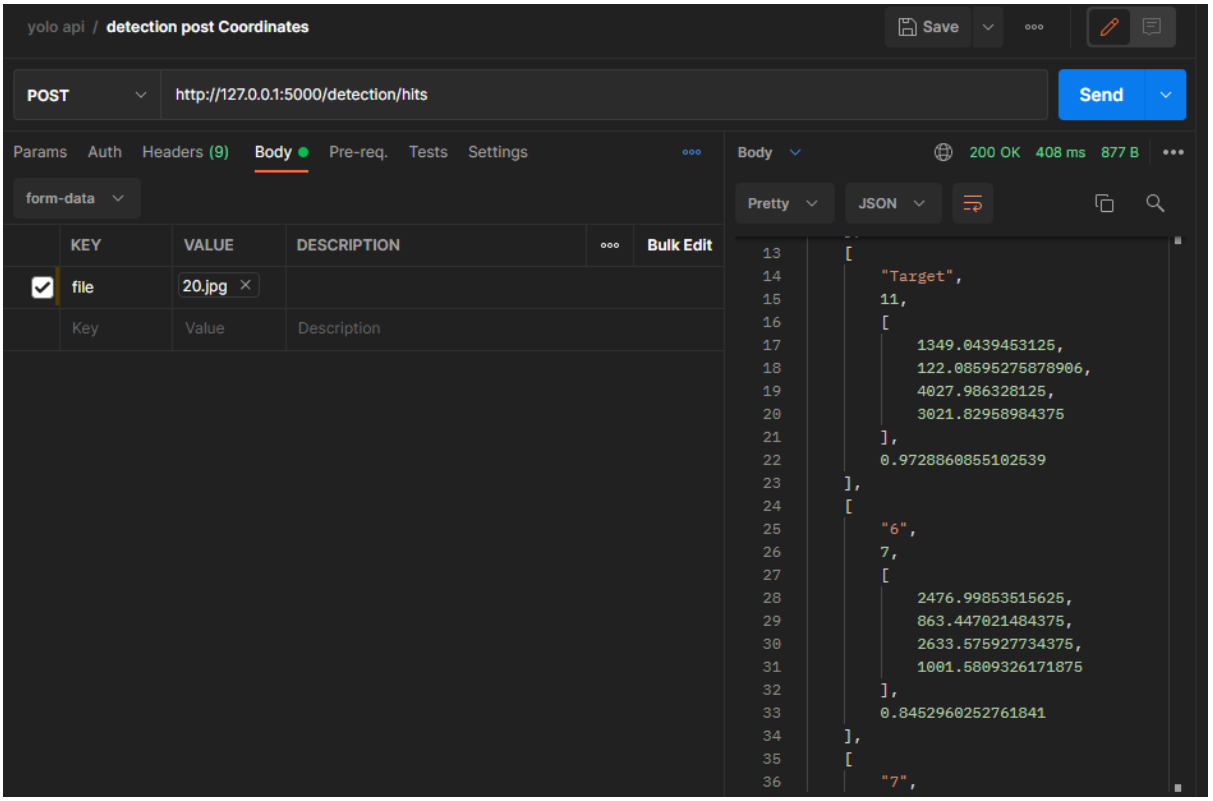

Figure 21. Result of a request to the API with coordinates of found objects.

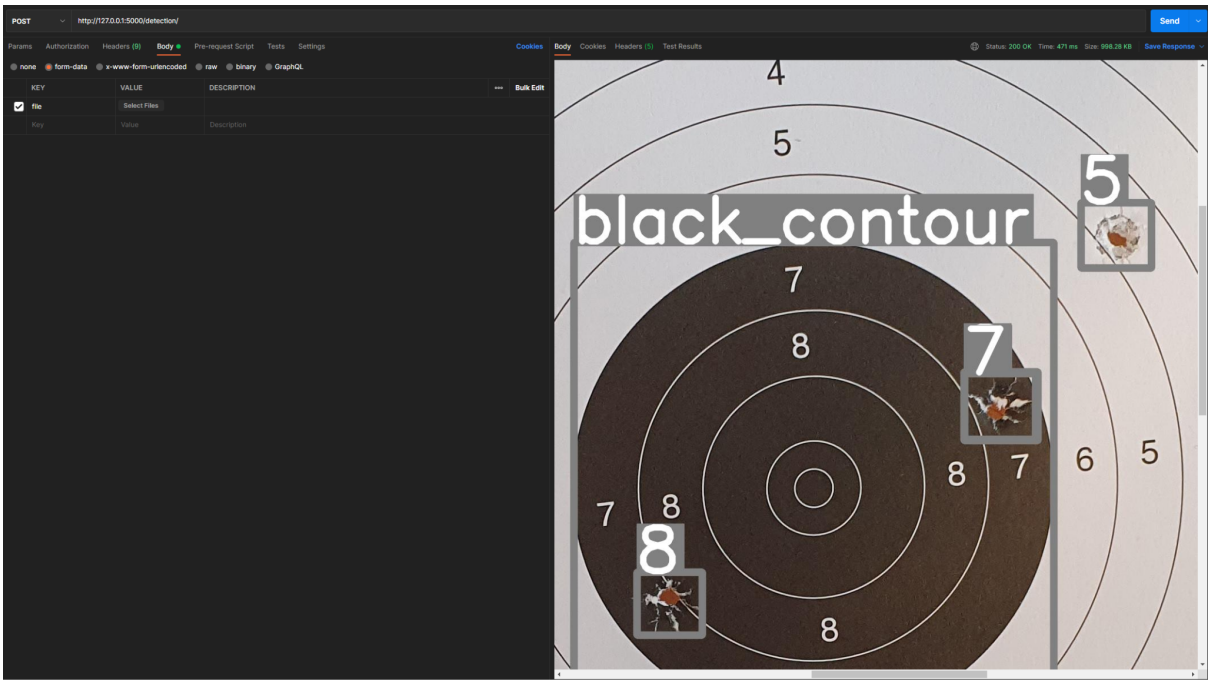

Figure 22. Result of a request to our API endpoint with the results.

After these endpoints had been constructed I decided to use my old shootsoft generator as the basis for the new generator. The final version of this new generator can be found in the shootsoftapiyolo.ipynb file. I started out by using all of our new

images that we collected by going to the shooting range to crop out each shot. In the end I had around 37 white shots and about 25 black shots. I thought for now that this would be enough. The shots in the generator are randomly rotated and the aspect ratio is also changed according to the angle of the photo.

To send requests to our API I made use of the following snippet (Figure 23).

```
def get_target(filename):
    url = 'http://127.0.0.1:5000/detection/hits'
    files = {'file': open(f'shootanalysisdataset/imagesnoshots/{filename}', 'rb')}
    response = requests.post(url, files=files)
    response_dict = json.loads(response.text)
    response_dict

    matches = [x for x in response_dict if x[0] == "Target"]

    contour_matches = [x for x in response_dict if x[0] == "black_contour"]

    black_contour = contour_matches[0]
    target = matches[0]

    return target, black_contour
```

Figure 23. Code snippet of the shootsoft generator for getting the target location

After the shots were generated I then had to manually go through the images to check whether they were actually realistic enough. Some of the generated images had to be filtered out because the black shots were not done properly because of the computer vision. Some images also had shots through my thumb.

After this I had to make a second second function that creates a XML file for each image with all of the detected shots. This XML file uses some type of uniform annotation used by roboflow. Figure 24 shows the code snippet and the final result.

```python
def get_shoot_values(filename):
    url = 'http://127.0.0.1:5000/detection/hits'
    files = {'file': open(f'shootanalysisdataset/generatedcleantargetdataset/{fi
    response = requests.post(url, files=files)
    response_dict = json.loads(response.text)
    response_dict


    return response_dict

def create_annotation(filename):

    values = get_shoot_values(filename)

    image = cv2.imread(f'shootanalysisdataset/generatedcleantargetdataset/{filen

    #removes the .jpg
    filename = filename.split(".")
    filename = filename[0]

    f = open(f"shootanalysisdataset/annotations/{filename}.xml", "w+")
    f.write("<annotation>")
    f.write("<folder></folder>")
    f.write(f"<filename>{filename}.jpg</filename>")
    f.write(f"<path>{filename}.jpg</path>")
    f.write(f"<source>roboflow.ai</source>")
    f.write(f"<size>")
    f.write(f"<width>{image.shape[1]}</width>")
    f.write(f"<height>{image.shape[0]}</height>")
    f.write(f"</size>")
    f.write(f"<depth>3</depth>")
    f.write(f"<segmented>0</segmented>")

    for detected_object in values:
        print(detected_object)

        #removes confidence below 0.4
        if detected_object[3] < 0.4:
            continue

        f.write(f"<object>")

        #EXTRA CHECK TO ADD BULLET_
        if detected_object[0] == "Target":
            f.write(f"<name>{detected_object[0]}</name>")
        else:
            f.write(f"<name>Bullet_{detected_object[0]}</name>")

        f.write(f"<pose>Unspecified</pose>")
        f.write(f"<truncated>0</truncated>")
        f.write(f"<difficult>0</difficult>")
        f.write(f"<occluded>0</occluded>")
        f.write(f"<bndbox>")
        f.write(f"<xmin>{detected_object[2][0]}</xmin>")
        f.write(f"<xmax>{detected_object[2][2]}</xmax>")
        f.write(f"<ymin>{detected_object[2][1]}</ymin>")
        f.write(f"<ymax>{detected_object[2][3]}</ymax>")
        f.write(f"</bndbox>")
        f.write(f"</object>")
    f.write("</annotation>")
```

```xml
<annotation>
    <folder></folder>
    <filename>1_2036-14-04-2023_API_v2.jpg</filename>
    <path>1_2036-14-04-2023_API_v2.jpg</path>
    <source>roboflow.ai</source>
    <size>
        <width>4032</width>
        <height>3024</height>
    </size>
    <depth>3</depth>
    <segmented>0</segmented>
    <object>
        <name>Target</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <occluded>0</occluded>
        <bndbox>
            <xmin>1374.7132568359375</xmin>
            <xmax>2946.18017578125</xmax>
            <ymin>651.484130859375</ymin>
            <ymax>2238.97900390625</ymax>
        </bndbox>
    </object>
    <object>
        <name>black_contour</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <occluded>0</occluded>
        <bndbox>
            <xmin>1806.0406494140625</xmin>
            <xmax>2416.7353515625</xmax>
            <ymin>1169.4739990234375</ymin>
            <ymax>1792.7005615234375</ymax>
        </bndbox>
    </object>
    <object>
        <name>Bullet_10</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <occluded>0</occluded>
        <bndbox>
            <xmin>2098.0791015625</xmin>
            <xmax>2152.40966796875</xmax>
            <ymin>1531.8922119140625</ymin>
            <ymax>1578.9530029296875</ymax>
        </bndbox>
    </object>
    <object>
        <name>Bullet_7</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <occluded>0</occluded>
        <bndbox>
            <xmin>1841.3668212890625</xmin>
            <xmax>1893.4202880859375</xmax>
            <ymin>1603.1572265625</ymin>
            <ymax>1652.33984375</ymax>
        </bndbox>
    </object>
</annotation>
```

Figure 24. Code snippet for generating the xml files.

All that was needed now was a way to send the image and XML file to roboflow. Figure 25 is the final result of that. The API key has been removed for obvious reasons.

```python
import os
import requests
import base64
import io
from PIL import Image


def send_image_to_roboflow(filename):

    #removes the .jpg
    filename_no_extension = filename.split(".")[0]

    # Load Image with PIL
    image = Image.open(f'shootanalysisdataset/generatedcleantargetdataset/{filenam

    # Convert to JPEG Buffer
    buffered = io.BytesIO()
    image.save(buffered, quality=90, format="JPEG")

    # Base 64 Encode
    img_str = base64.b64encode(buffered.getvalue())
    img_str = img_str.decode("ascii")

    # Construct the URL
    upload_url = "".join([
      "https://api.roboflow.com/dataset/project-oqaxk/upload",
      "?api_key=",
      f"&name={filename}",
      "&split=train"
    ])

    # POST to the API
    r = requests.post(upload_url, data=img_str, headers={
      "Content-Type": "application/x-www-form-urlencoded"
    })

    # Output result
    print(r.json())

    img_id = r.json()['id']

    annotation_filename = os.path.splitext(filename)[0]+'.xml'
    print(annotation_filename)

    # Read Annotation as String
    annotation_str = open(f"shootanalysisdataset/annotations/{filename_no_extensio


    # Construct the URL
    upload_url = "".join([
    "https://api.roboflow.com/dataset/project-oqaxk/annotate/" + img_id,
    "?api_key=",
    "&name=", f"{filename_no_extension}.xml"
    ])

    # POST to the API
    r = requests.post(upload_url, data=annotation_str, headers={
    "Content-Type": "text/plain"
    })

    # Output result
    print(r.json())
```

Figure 25. Snippet of code to send the xml and image to roboflow.

In total I generated about 80 images in week 6. After the images were sent I did have to manually correct all images where the annotations were incorrect. Figure 26 shows the confusion matrix of the model I generated with this dataset.

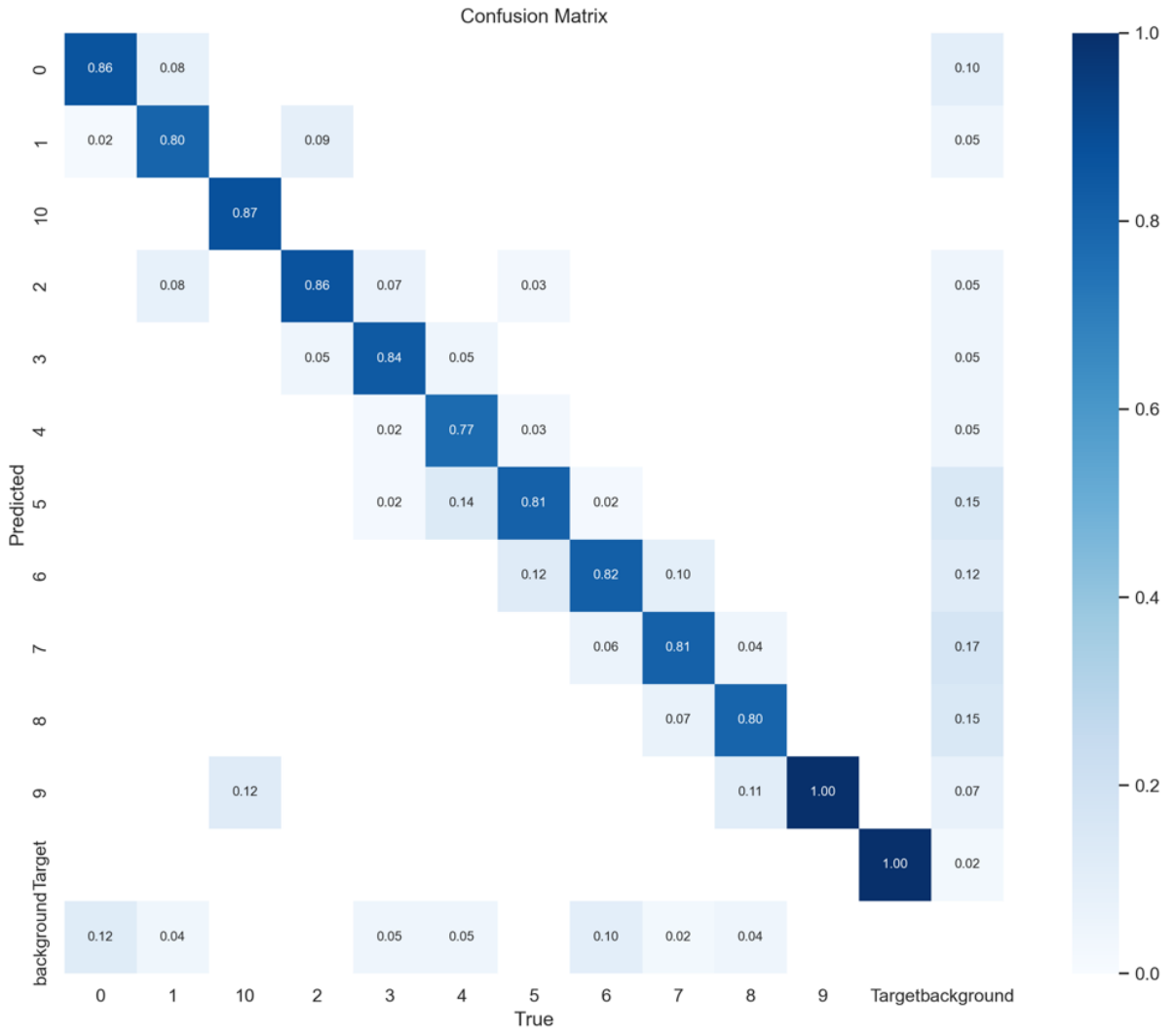

Figure 26. Confusion matrix model week 6

Week 8

I started with week 8 by making the design for the Shootsoft application in Figma. Thijs send us some inspiration of how a possible design for the front-end application could look. I've carefully examined what Thijs sent and then decided to design something which does not resemble his design at all. The design is part of this document and can be found here.

In week 8 I also wanted to generate more images for the final dataset. The one point where the generation of shooting cards still struggles was with finding the black contour. This was one of the only tasks still done by computer vision. I first of all tried to only find the contours within the detected target as a sort of hybrid solution. The only problem is that it still sometimes resulted in misshaped black contours. Since the target is detected extremely accurately in our model I decided to also annotate all of the black contours. When doing this I also redistributed the dataset for training, validation and testing.

Before editing the generator and generating more images I wanted to see what the result would be with this new validation set. I also upped the image quality from 840 pixels to 1008 which is thrice as small as the smallest dimension of most of our images. The results of this model were varied. Figure 27 shows the confusion matrix of this model. This new model was very accurate in some instances but also very inaccurate in some other areas. The nine had for instance only a 58% percent accuracy. This was quite shocking and I decided to check out the validation set. It turned out that the validation set had quite a lot of annotation mistakes. The 8 and 9 were mixed up a lot and there were unnecessarily large margins around the detected bullets. I went through the entire validation set to do a double check. After this was done I realized It was probably wise to go over the entire dataset again to check all of the images. Annotating is quite a braindead job and this way mistakes can of course sometimes slip in. This can however if it happens often enough have a negative impact on the result.

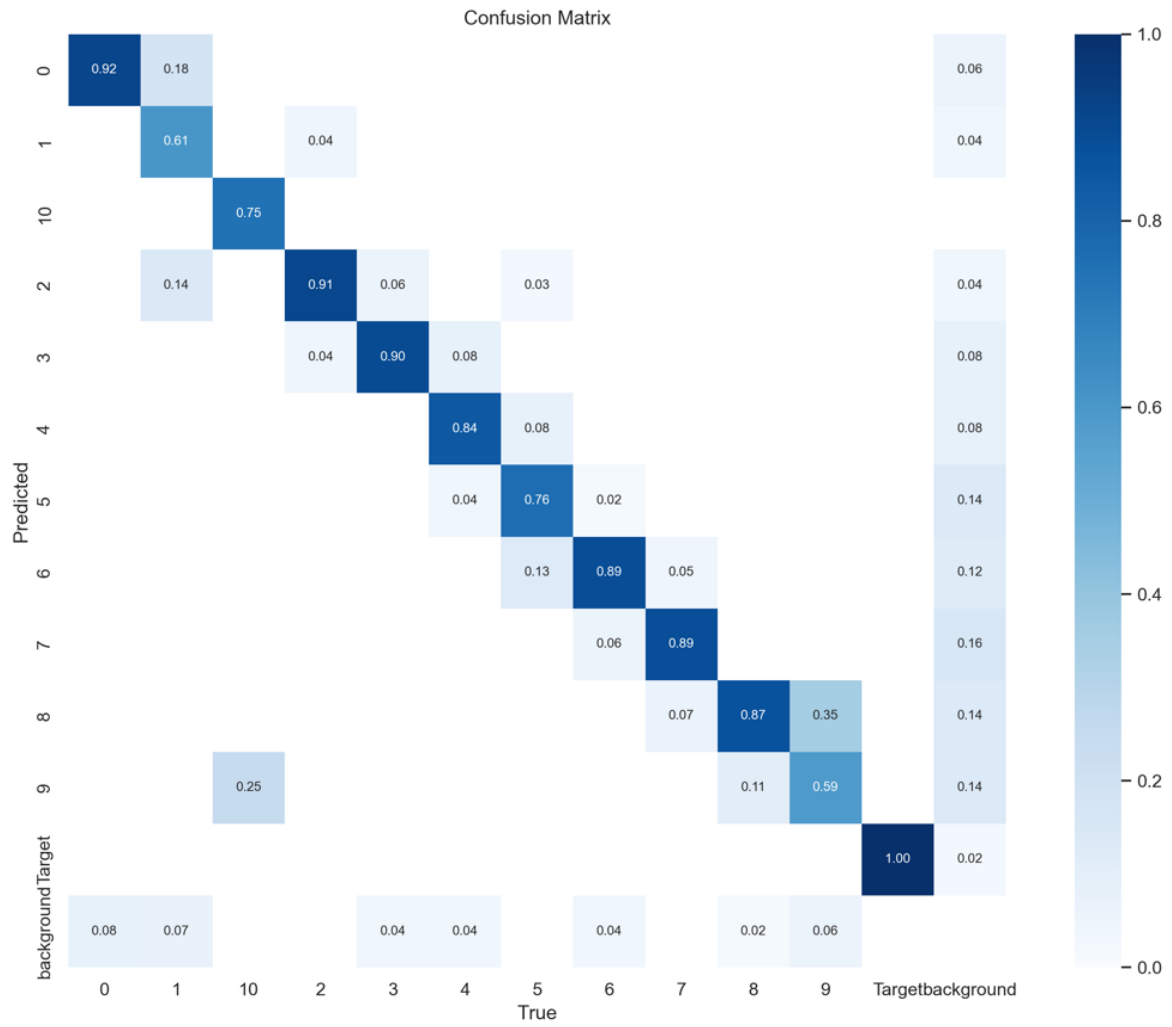

Figure 27 Confusion matrix of the new model

After I had checked the dataset I went back to generate more images. First of all I had to edit the generator to use the detected black contour from the model. This resulted in a much more reliable result and more accurately generated images. Figure 28 shows the result of such an image.

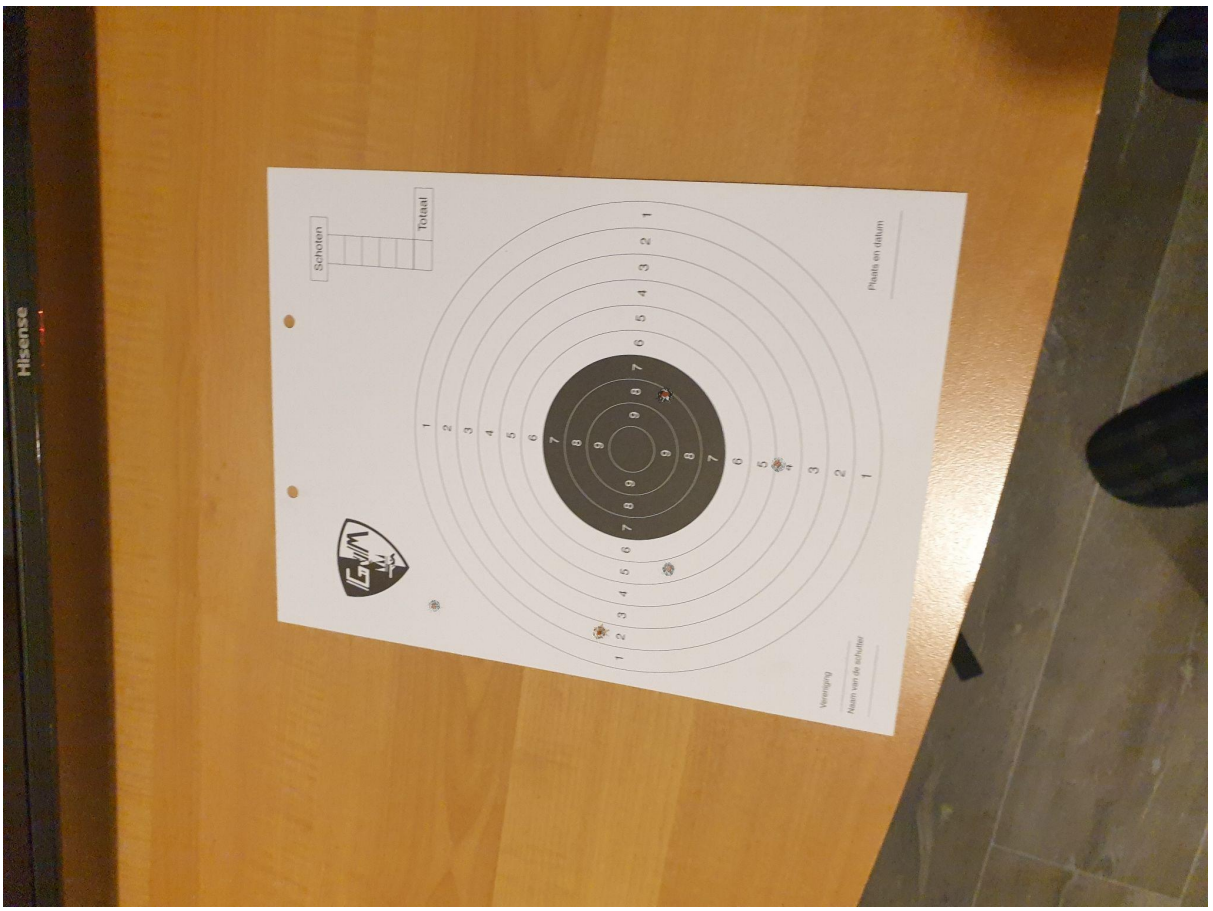

Figure 28 Generated image example taken from the shootsoft generator

I decided to generate 180 extra images and add these to the dataset. Afterwards I still had to edit every single image by hand to correct the detections and make sure that the classes were accurate. When this was done I noticed with the roboflow health check that some classes like 9 or 10 were underrepresented. This is ofcourse logical if you randomly generate data since these two classes have the smallest surface. I fixed this by generating images that only had shots in the black contour. Getting this contour through the model had made this possible. Figure 29 is an example of this. I generated about 80 images in total to get the count of all classes a bit closer together. After correcting the annotations of these 80 images I decided to train another model. Figure 30 is the final result of this model. The lowest accuracy score on this model is 84% and I was positively surprised.

I finished week 7 by working on my log and writing parts of the TFGD.

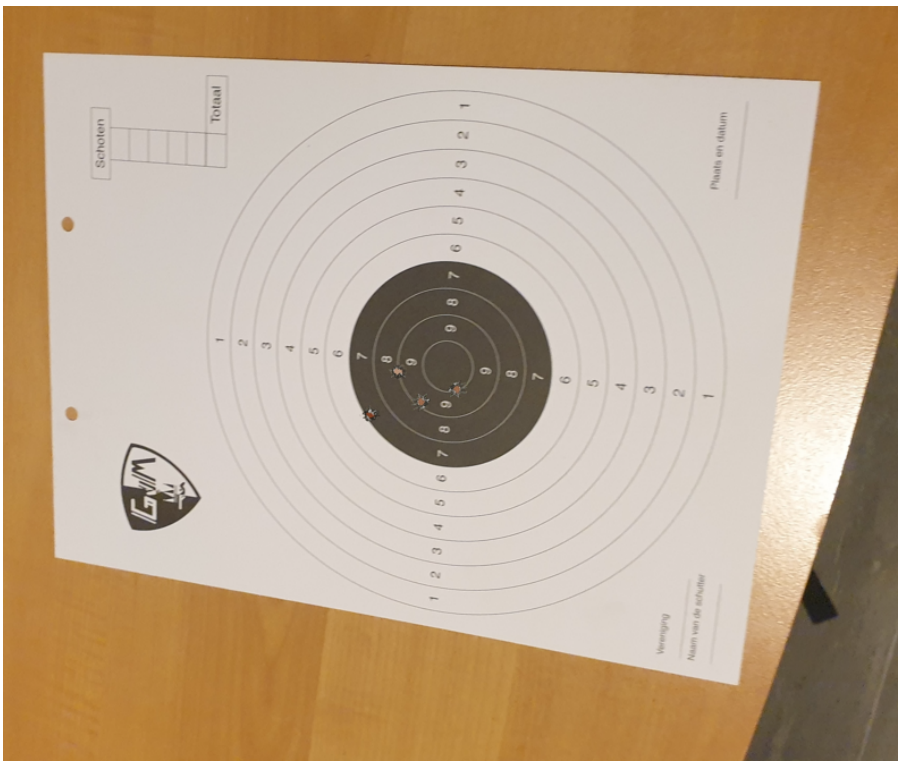

Figure 29. Generated images with only shots in the 7,8,9,10

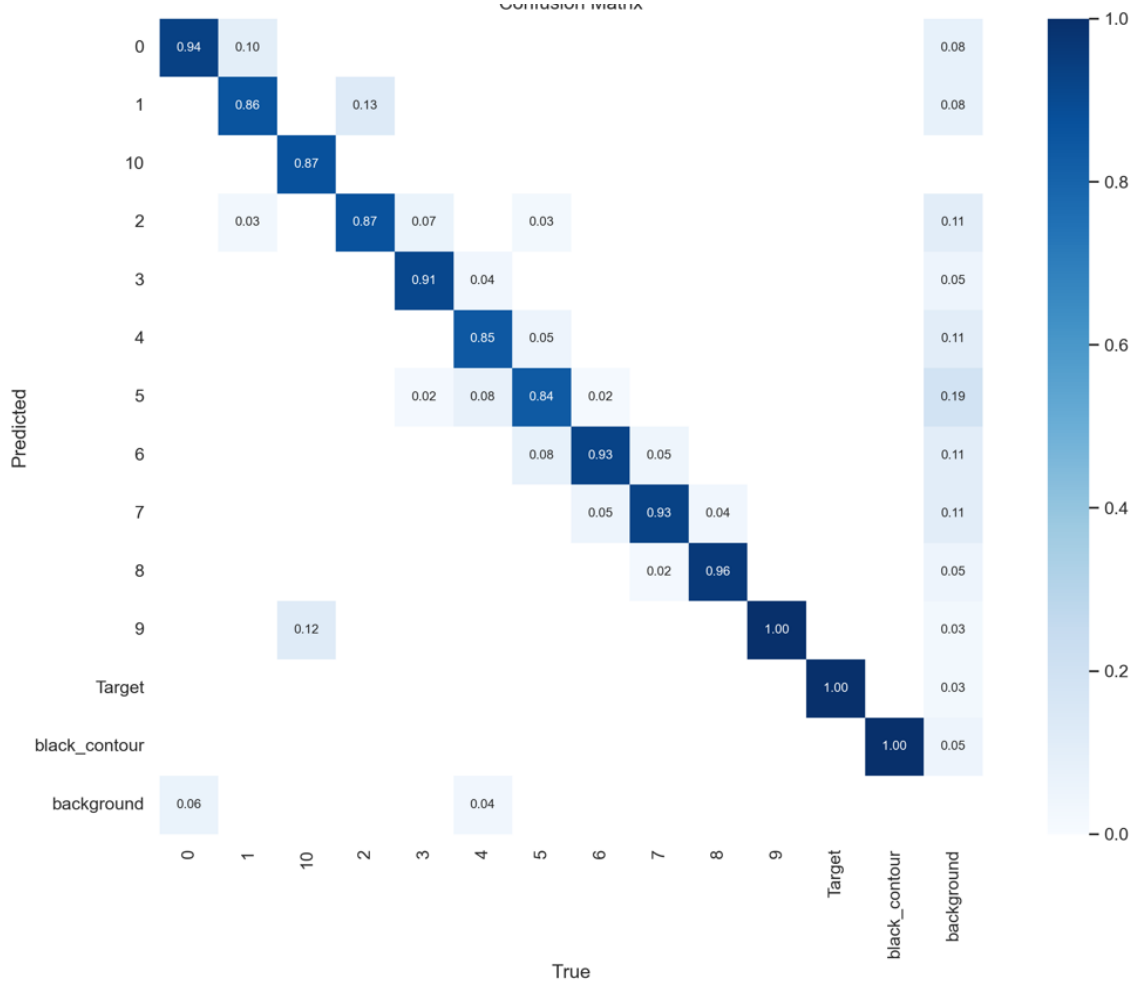

Figure 30. Confusion matrix of the model trained on the new dataset.