Scrabble calculator

1. Problem description

We want to built an automatic scrabble calculator. For each round the software outputs the positions for each new letter, the predicted letters for those positions and the round score. We divide this into 3 tasks: Task 1 (positions), Task 2 (letter classification), Task 3 (score).

1.0. Data description

Training data consist of 100 images with ground_truth named i_j.jpg, i_j.txt respectively, where i start from 1 to 5 and j starts from 1 to 20. In total we have 5 games each with 20 rounds.

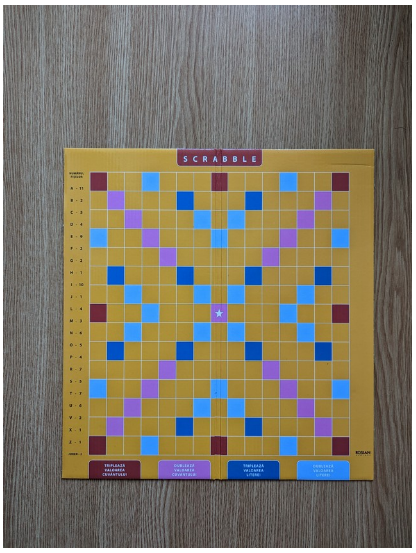


Figure 1: Empty board

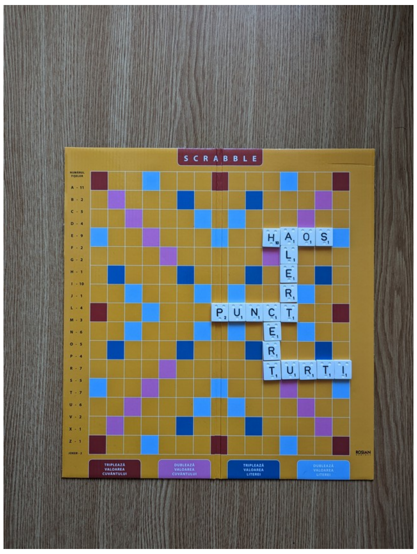


Figure 2: Board with letters

2. Extract puzzle from input image

First of all we need to crop the board. Using *Filtrarea culorilor applications and HSV format we can extract values for low(lh, ls, lv) and high(hh, hs, hv) to create a mask that indicates the red spots in the image (Note: we do not need the whole board, just the puzzle inside it i.e. the places where letters are attached). For low(154, 54, 0) and high(255, 255, 255) the output is:*

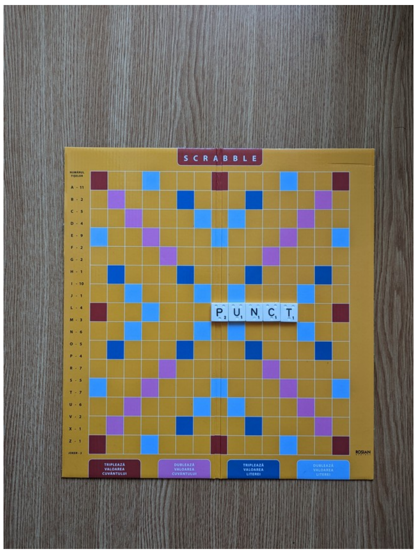


Figure 3: Original image

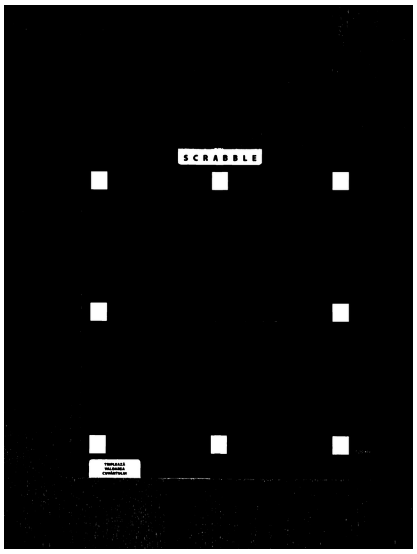


Figure 4: Mask

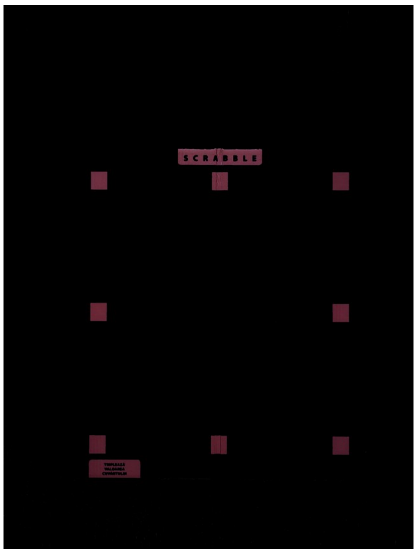


Figure 5: Resulted image

There are some white holes on the mask that need to be eliminated. We can use morphological operations to fill them with black. Using erosion with a kernel size of 3x3 we got the following mask.

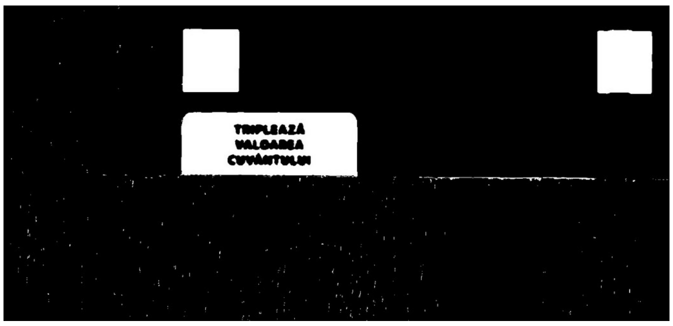


Figure 6: Original mask (zoom in)

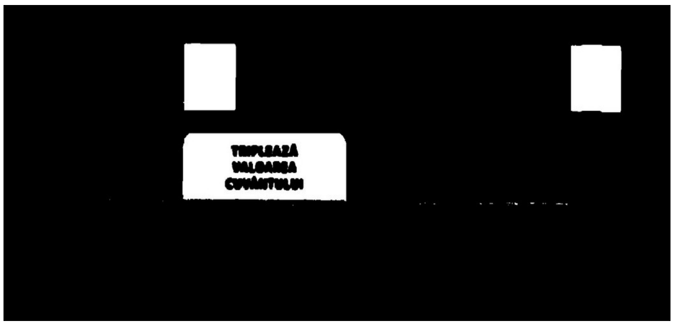


Figure 7: Eroded mask (zoom in)

This way we have a cleaner mask to apply Canny edge detection for 'red' square extraction. To find the right values for threshold1 and threshld2 parameters we compute the *mean* of the mask, then compute the values $l = 0.66 * mean$ and $u = 1.66 * mean$. We use l as threshold1 and u as threshold2. The following image are the edges from out mask.

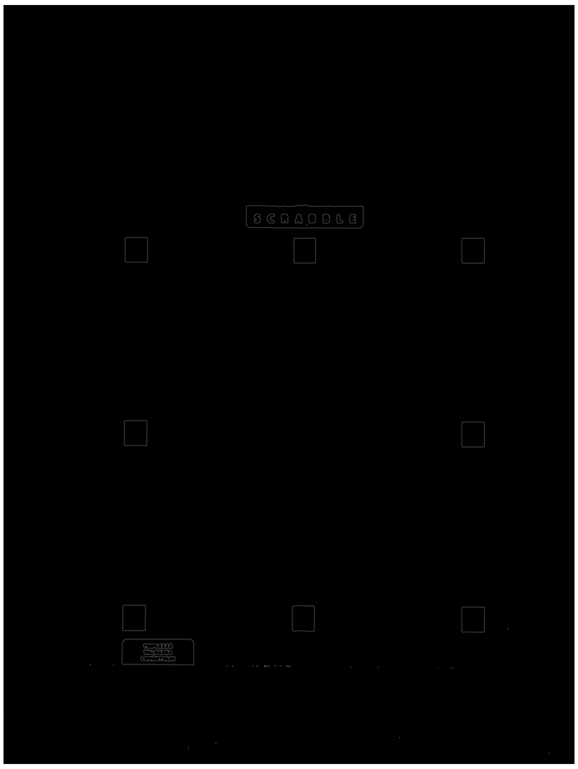


Figure 8: Canny edge detection (whole image)

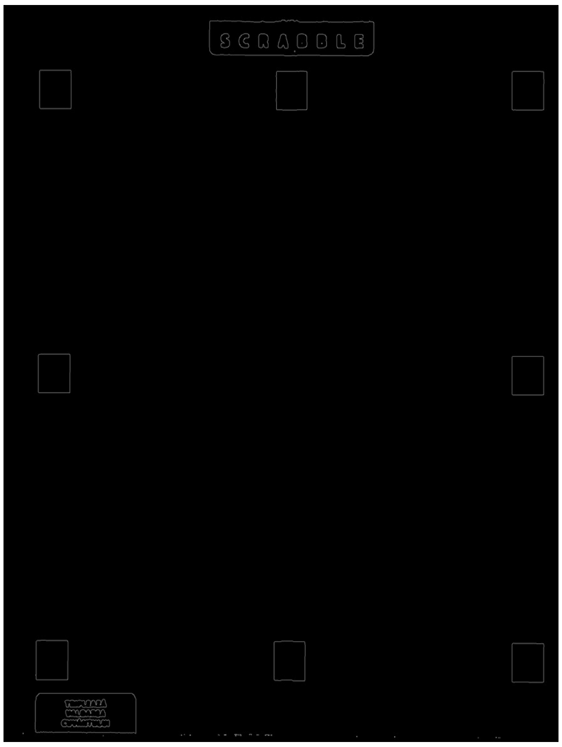
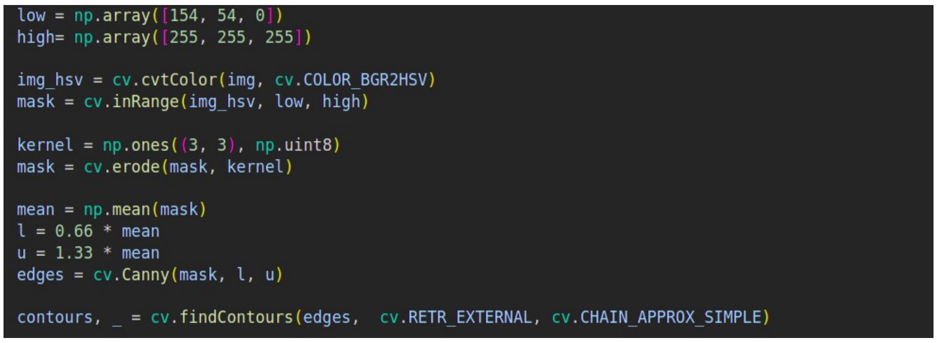


Figure 9: Canny edge detection (zoom in)

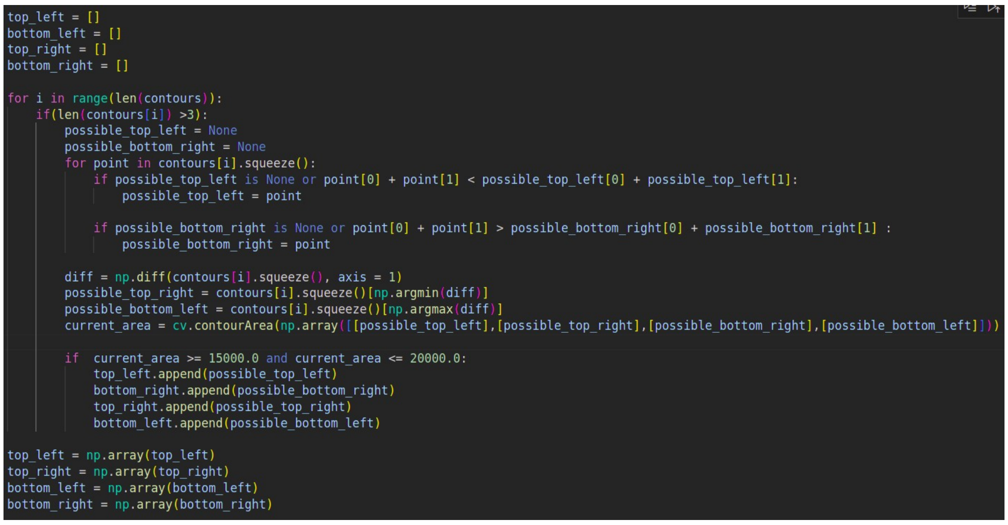
Next we can find the contours using open cv function cv2.findContours. The following screenshot shows the code until here.



```
low = np.array([154, 54, 0])
high= np.array([255, 255, 255])

img_hsv = cv.cvtColor(img, cv.COLOR_BGR2HSV)
mask = cv.inRange(img_hsv, low, high)

kernel = np.ones((3, 3), np.uint8)
mask = cv.erode(mask, kernel)

mean = np.mean(mask)
l = 0.66 * mean
u = 1.33 * mean
edges = cv.Canny(mask, l, u)

contours, _ = cv.findContours(edges,  cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)
```

Figure 10: Finding the contours of the puzzle

With these contours we can extract the corners of all 8 red squares. Experimentally we can see that area of one red square is between 15000 and 20000. With this information coordinates can be extracted for all four corners of all 8 squares. The output and the code are present bellow (Note: Figure 11.2 is created only for this documentation. The final implementation do not create the image).



```python
top_left = []
bottom_left = []
top_right = []
bottom_right = []

for i in range(len(contours)):
    if(len(contours[i]) >3):
        possible_top_left = None
        possible_bottom_right = None
        for point in contours[i].squeeze():
            if possible_top_left is None or point[0] + point[1] < possible_top_left[0] + possible_top_left[1]:
                possible_top_left = point

            if possible_bottom_right is None or point[0] + point[1] > possible_bottom_right[0] + possible_bottom_right[1] :
                possible_bottom_right = point

        diff = np.diff(contours[i].squeeze(), axis = 1)
        possible_top_right = contours[i].squeeze()[np.argmin(diff)]
        possible_bottom_left = contours[i].squeeze()[np.argmax(diff)]
        current_area = cv.contourArea(np.array([[possible_top_left],[possible_top_right],[possible_bottom_right],[possible_bottom_left]]))

        if  current_area >= 15000.0 and current_area <= 20000.0:
            top_left.append(possible_top_left)
            bottom_right.append(possible_bottom_right)
            top_right.append(possible_top_right)
            bottom_left.append(possible_bottom_left)

top_left = np.array(top_left)
top_right = np.array(top_right)
bottom_left = np.array(bottom_left)
bottom_right = np.array(bottom_right)
```

Figure 11.1: Detection of the red squares (code)

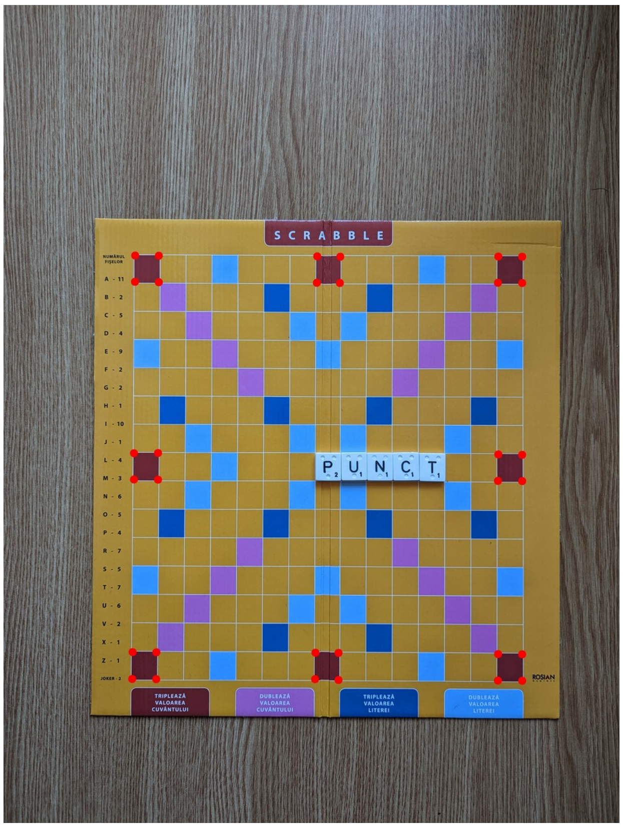

SCRABBLE
PUNCT
TRIPLEAZĂ VALOAREA CUVÂNTULUI
DUBLEAZĂ VALOAREA CUVÂNTULUI
TRIPLEAZĂ VALOAREA LITEREI
DUBLEAZĂ VALOAREA LITEREI

Figure 11.2: Detection of the red squares (image)

For the puzzle we need to extract the area between the corners showed in Figure 12.

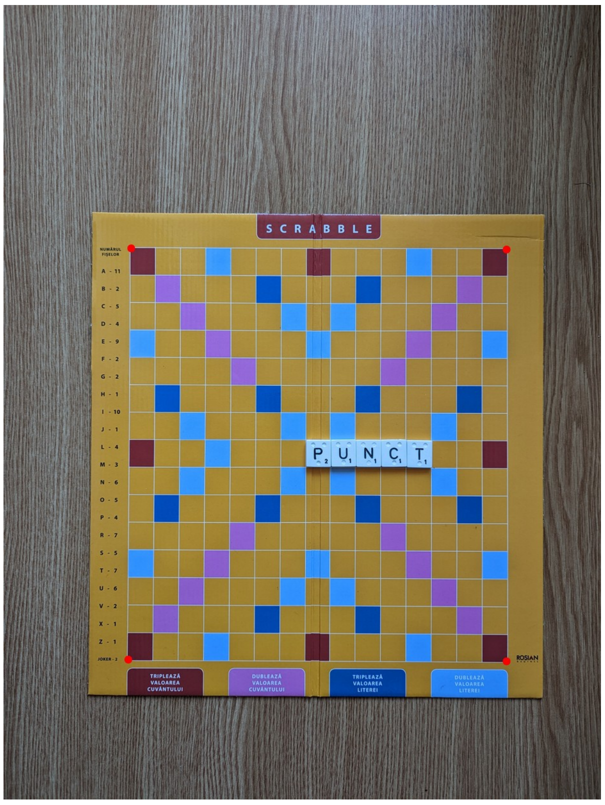
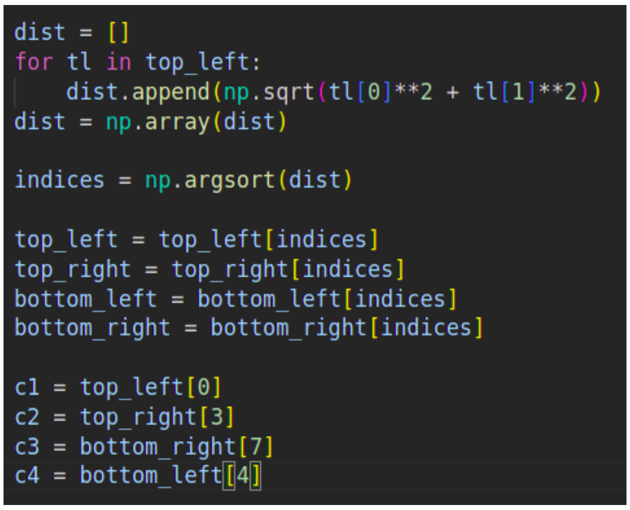


Figure 12: Corners of the puzzle

To do this we can use the Euclidian distances of the red squares. Based on this distance sort (ascending) the red squares coordinates. Then take the first red square top left as corner 1, the forth red square top right as corner 2, the eighth red square bottom right as corner 3 and the fifth red square bottom left as corner 4 (Note: The corners are indexed clock wise starting from top left red square of the puzzle).



```
dist = []
for tl in top_left:
    dist.append(np.sqrt(tl[0]**2 + tl[1]**2))
dist = np.array(dist)

indices = np.argsort(dist)

top_left = top_left[indices]
top_right = top_right[indices]
bottom_left = bottom_left[indices]
bottom_right = bottom_right[indices]

c1 = top_left[0]
c2 = top_right[3]
c3 = bottom_right[7]
c4 = bottom_left[4]
```

Figure 13: Finding the four corners

Finally, we warp the area delimited by those corners into a 1500x1500 image.

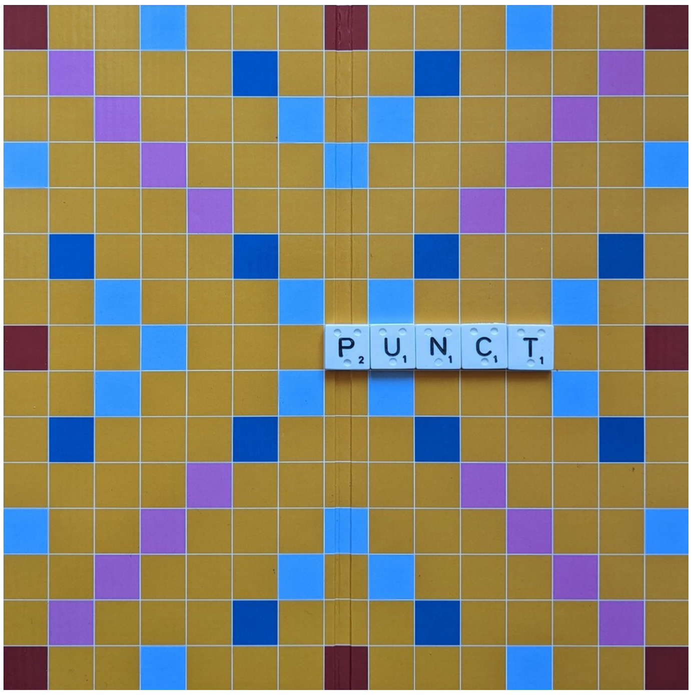


Figure 14: Warped puzzle

Because every shot is taken under the same conditions (the only things that change are letters and brightness) we can do this process only for the first round, then remember the corners. This way we save computational time and solve the problem that may occur when a player places a letter on one of the four red squares from the corners.

In the implementation this process is done by the *get_puzzle(img, corners=[])* function.

3. Extract information from puzzle

In this section we will discuss the process of getting the information needed for our application, i.e. the positions, the letter for all positions and the score. At every first round of a game (j = '01') the corners of the puzzle are computed. Then for every round we get an warped image with the puzzle like the one in Figure 14. We set two list with horizontal lines and vertical lines respectively. The lines are ilustrated in the Figure 15.

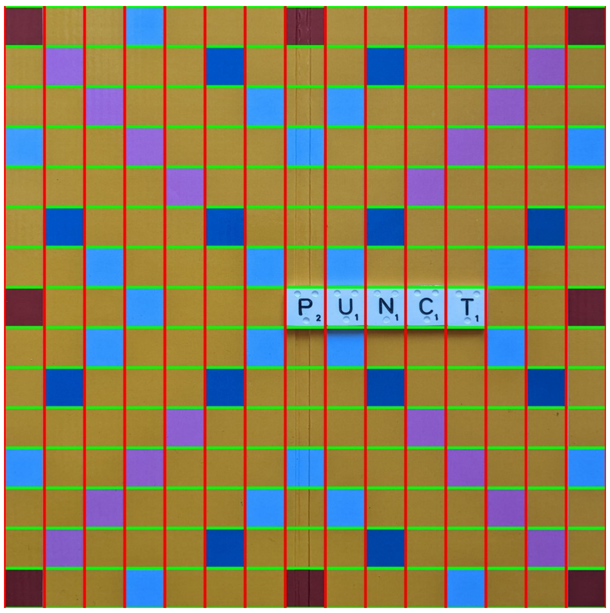


Figure 15: Puzzle with lines

Iterate over those two lists like in a matrix and at each step get the patch between the lines (one square in the puzzle). With the patch we create a binary mask to highlight the letter and apply erosion to get a more full image. Figure 16 shows this process applied on the whole puzzle, without erosion.

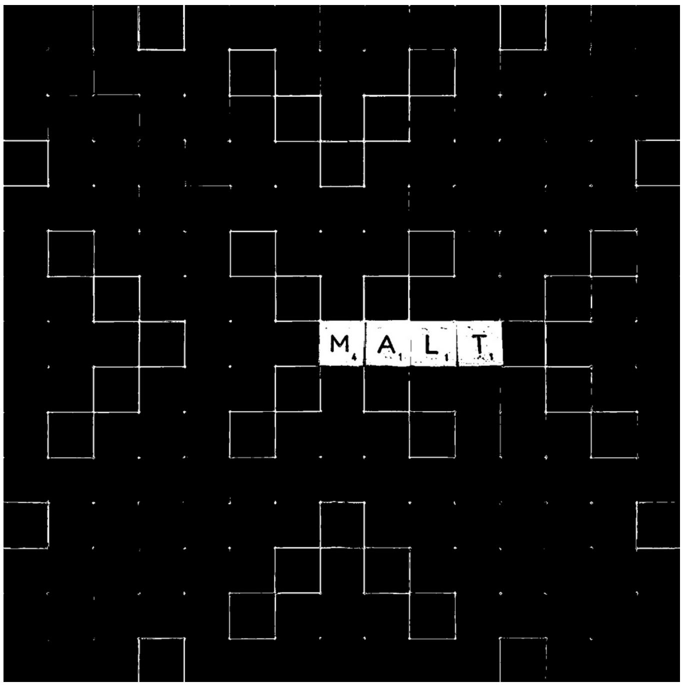


Figure 16: Mask on the whole puzzle (without erosion)

Next, on eroded mask get the floor of the mean of the patch. If the mean is greater than 10, we have a letter. Write down into a string the position of the letter, and if this position is new in the round, then predict the letter. At the end of this search for letters we compute the score of the round, then write the results in the specific text file.

The letter classification is done by template matching. The Figure 17 shows some of the templates. Each template is 50x50 pixels and the patches are 80x80 pixels (Note: Joker is classified as 'W' and when the algorithm predicts 'W' we change the predicted string with '?').

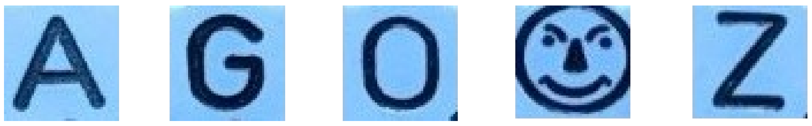
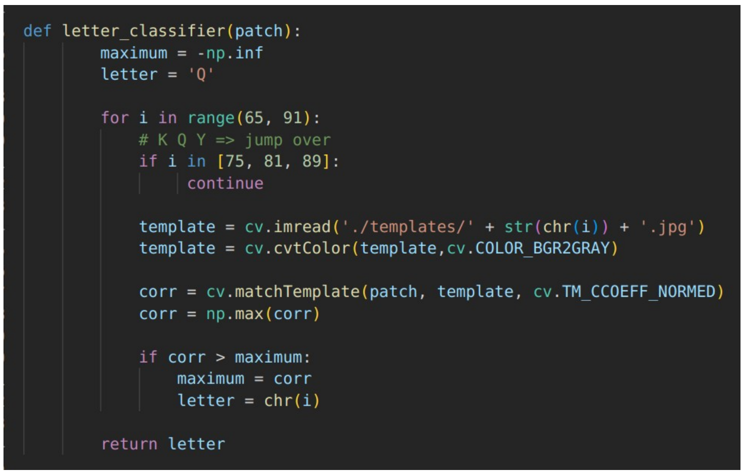


Figure 17: Example of templates

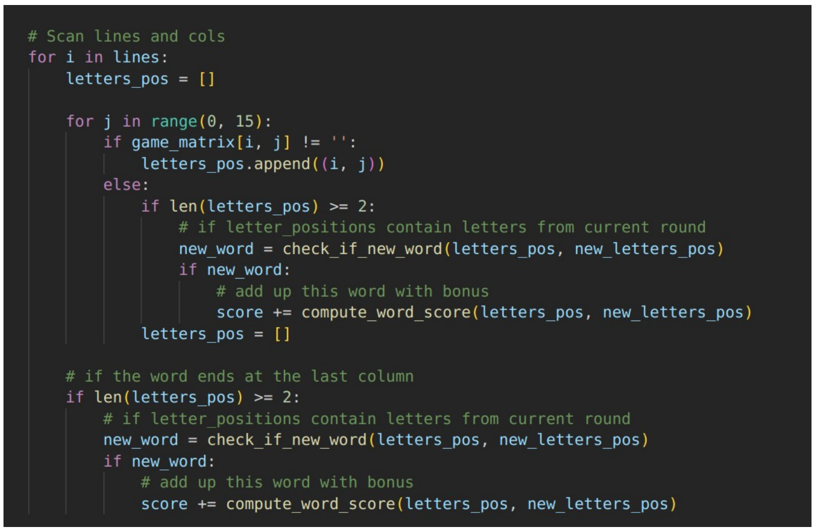
The code for classification is shown bellow.



```python
def letter_classifier(patch):
        maximum = -np.inf
        letter = 'Q'

        for i in range(65, 91):
            # K Q Y => jump over
            if i in [75, 81, 89]:
                continue

            template = cv.imread('./templates/' + str(chr(i)) + '.jpg')
            template = cv.cvtColor(template,cv.COLOR_BGR2GRAY)

            corr = cv.matchTemplate(patch, template, cv.TM_CCOEFF_NORMED)
            corr = np.max(corr)

            if corr > maximum:
                maximum = corr
                letter = chr(i)

        return letter
```

Figure 18: Code for letter classification

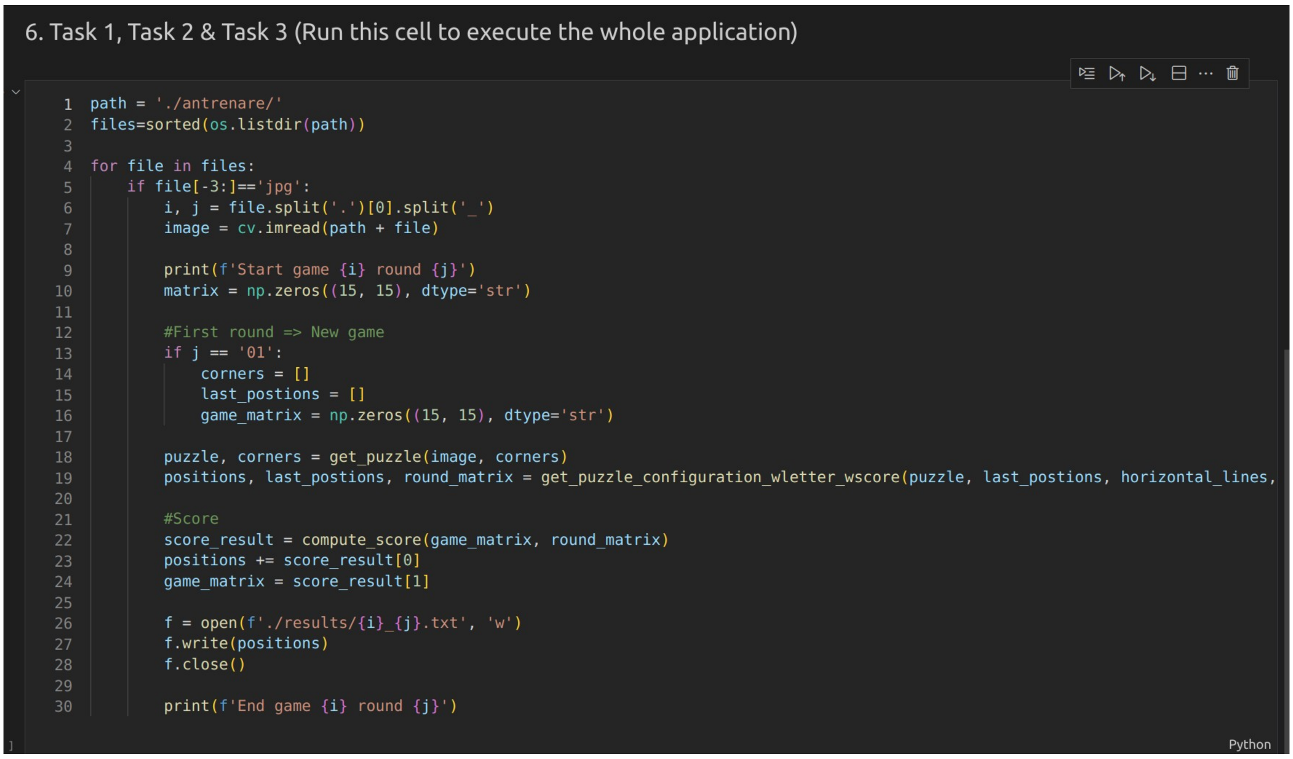
Now we need to compute the score. At each round the function that returns the positions of the letters, also returns a 15x15 matrix, let call it *round_matrix*, of strings with the predicted letters on that position and empty strings otherwise. We also have a *game_matrix* with the positions of all the letters from previous rounds. When we start to compute the score, first add the new letter in the *game_matrix*. Now we do a scan of the lines and columns where the new letters, i.e. letters from *round_matrix*, are placed and add to a list the positions of the possible new word. This scan is done on the *game_matrix* after we add the new letters. If we found a new word, then compute the score for that word and add it to the score of the whole round. A sample code is shown bellow.



```python
# Scan lines and cols
for i in lines:
    letters_pos = []

    for j in range(0, 15):
        if game_matrix[i, j] != '':
            letters_pos.append((i, j))
        else:
            if len(letters_pos) >= 2:
                # if letter_positions contain letters from current round
                new_word = check_if_new_word(letters_pos, new_letters_pos)
                if new_word:
                    # add up this word with bonus
                    score += compute_word_score(letters_pos, new_letters_pos)
            letters_pos = []

    # if the word ends at the last column
    if len(letters_pos) >= 2:
        # if letter_positions contain letters from current round
        new_word = check_if_new_word(letters_pos, new_letters_pos)
        if new_word:
            # add up this word with bonus
            score += compute_word_score(letters_pos, new_letters_pos)
```

Figure 19: Code for scanning the lines of new letters

All the information is stored in a string then written into the text file. Bellow is a screenshot of this process.



6. Task 1, Task 2 & Task 3 (Run this cell to execute the whole application)

```python
path = './antrenare/'
files=sorted(os.listdir(path))

for file in files:
    if file[-3:]=='jpg':
        i, j = file.split('.')[0].split('_')
        image = cv.imread(path + file)

        print(f'Start game {i} round {j}')
        matrix = np.zeros((15, 15), dtype='str')

        #First round => New game
        if j == '01':
            corners = []
            last_postions = []
            game_matrix = np.zeros((15, 15), dtype='str')

        puzzle, corners = get_puzzle(image, corners)
        positions, last_positions, round_matrix = get_puzzle_configuration_wletter_wscore(puzzle, last_postions, horizontal_lines,

        #Score
        score_result = compute_score(game_matrix, round_matrix)
        positions += score_result[0]
        game_matrix = score_result[1]

        f = open(f'./results/{i}_{j}.txt', 'w')
        f.write(positions)
        f.close()

        print(f'End game {i} round {j}')
```

Figure 20: Main code for the application

6. Results on train data

The algorithm's performance is shown in the table bellow.

Metric	Task 1	Task 2	Task 3	All tasks
Accuracy	100%	99%	98%	-
Note	5 (out of 5)	1.98 (out of 2)	1.95 (out of 2)	8.93 (out of 9)

The problem is on image 1_04.jpg, where the algorithm predicts 'I' letter with 'J'. This affects two score computations that use this letter, because the score of 'I' is different than the score of 'J'.