University of Bucharest

Faculty of Mathematics and Computer Science

Concepts and Applications in Artificial Vision

Automatic score calculator for Scrabble

Chirobocea Mihail Bogdan

2022-2022

Content:

- Purpose and tasks

- Task 1 - finding letters position
  - Solving idea
  - Examples
  - How does the code work?

- Task 2 - classifying letters
  - Solving idea
  - Examples
  - How does the code work?

- Task 3 - score calculation
  - Solving idea
  - Examples
  - How does the code work?

Purpose and tasks

- ## Purpose

The purpose of this project is to implement an automatic score calculation system for the word game Scrabble. We are using a set of 100 photos from 5 games of Scrabble - 20 photos per game, one photo per turn - taken in a controlled environment. The board may slightly vary in position as well as the color balance, saturation, luminosity and shadows may slightly vary from one photo to another.

- ## Tasks
    - Task 1:

For each image, the algorithm must provide the piece positions of the letters that were placed on the board in the respective round.

    - Task 2:

For each piece position, the algorithm has to recognize the letter that is on that track

    - Task 3:

For each image, the algorithm has to calculate the score obtained by the player in the respective round.

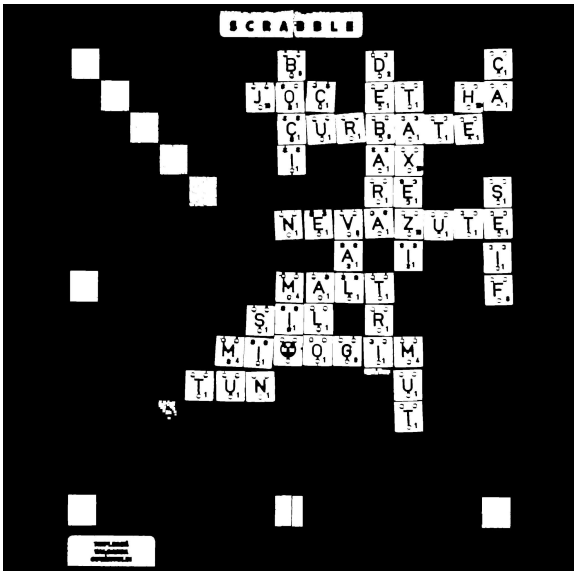
Task 1 - finding letters position

Solving idea

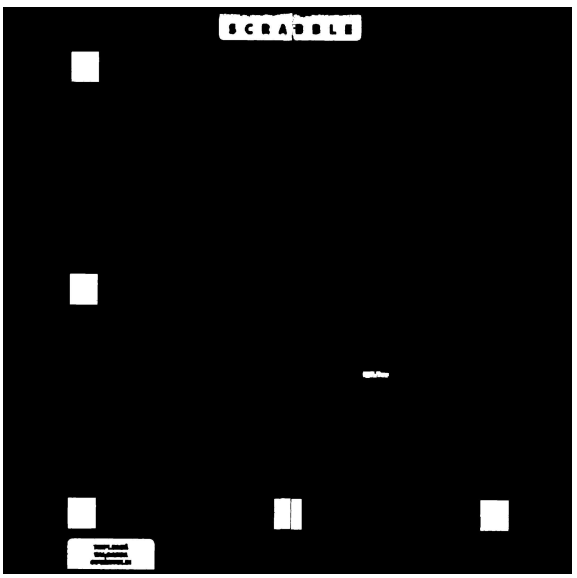
First, we want to cut the board from the image. In order to do that we are going to use a combination of HSV (Hue, Saturation, Value) masks. We can see that in the corners of the board there are red squares which are very easy to distinguish. So, we will make a red mask to select those corners, though on the board there are some other red figures that we would like to cut, we will fix this later. The red corners may be covered by pisces in some photos, that being said, we will also make a white mask. Combining these two masks we obtain this:

Original image (cropped)

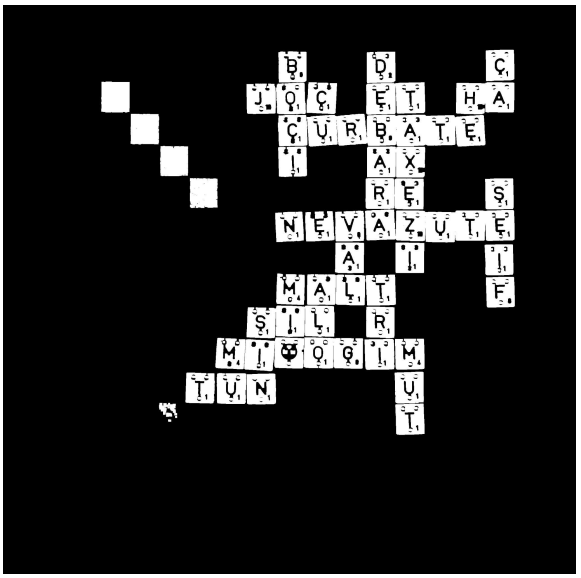
White+red mask



White mask



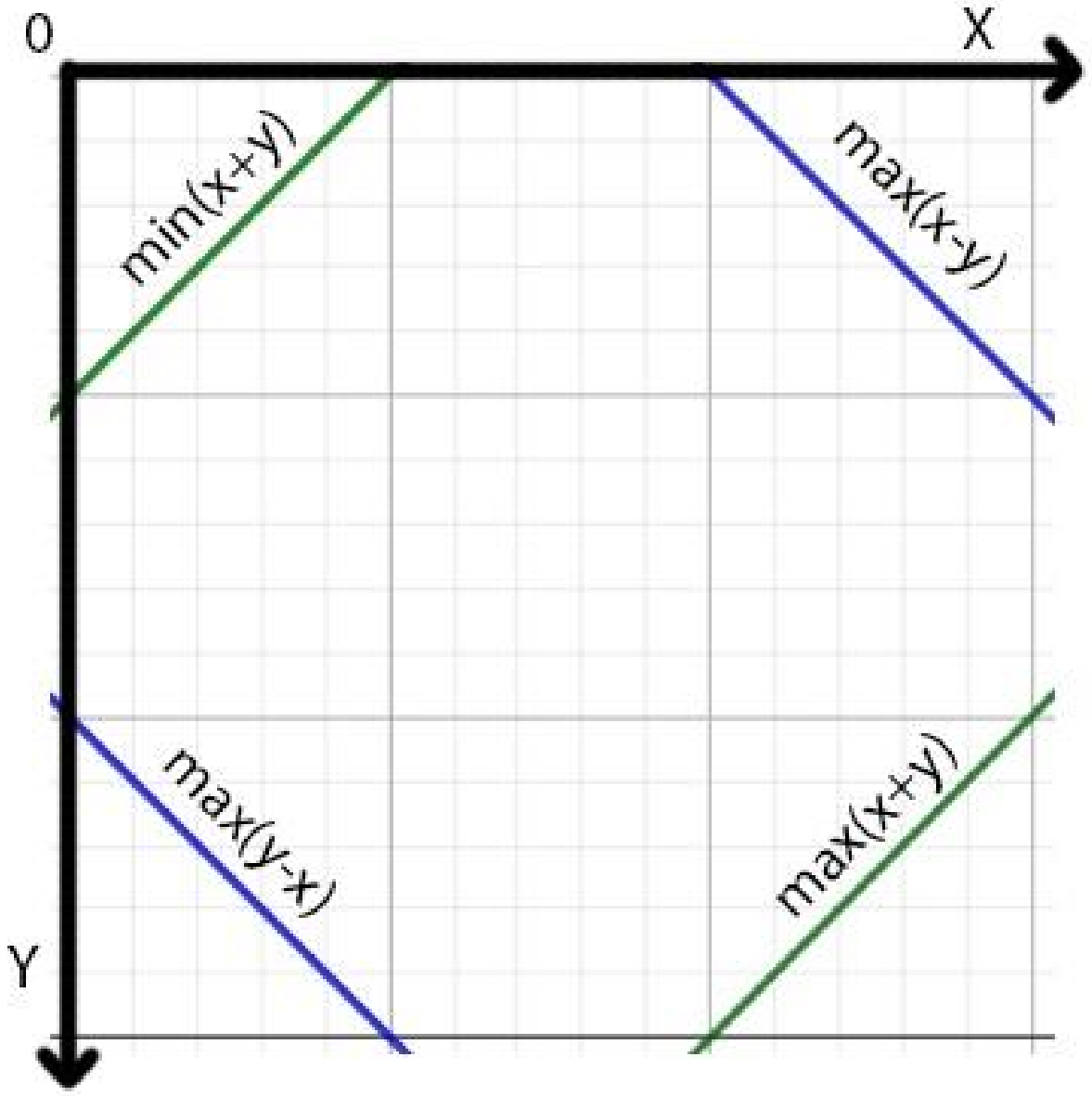
Red Mask



The two red borders, one on the top and one on the right-left, will be removed from the mask (see code).

Now, we want to find the corners of the board. We want to find the white dots with coordinates (x,y) which have:

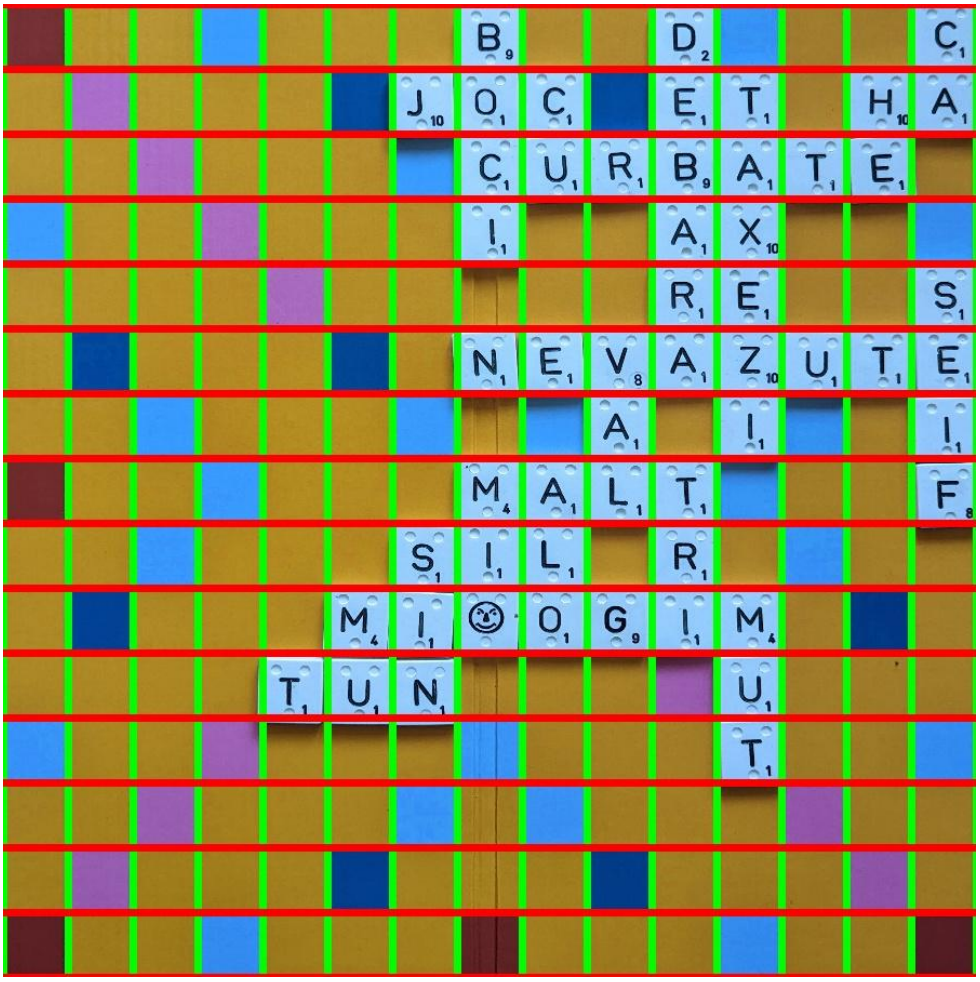
- The minimum x+y value, for top left corner
- The maximum x-y value, for top right corner
- The maximum x+y value, for bottom right corner
- The maximum y-x value, for bottom left corner



Keep in mind that the top left corner in a photo starts with coordinates (0,0).

Now that we have the corners of the board we can make a warp crop to remove distortion of the photo (see cod).

We are going to divide the board into 15 parts in length and 15 parts in width, thus we will obtain the 225 squares of the board game.

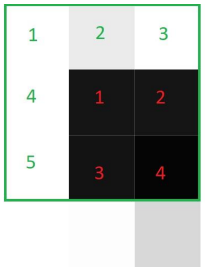


We will take squares small enough to remove the shadows on the pisces (made by other pisces) and big enough to keep the letter (or at least a part of it) in the square.

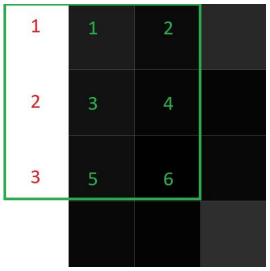
We can make a mask to fit only the darkest tone of the letters. Based on that mask made on each square we will find out if there is a letter or not.

For this mask we will apply a median blur to remove possible death pixels or other aberrations that could be dark enough for the initial mask. Some letters have a thickness of only 3 pixels, so even the smallest median blur will strongly affect the brightness value of that letter. We will resize the path on both axis with an x2 ratio. In that way, a death pixel will come out to 4 pixels, which is still small enough to be affected by the median blur (3x3 filter size).

Dead pixel (x2)



Edge letter (x2)



How does the code work?

As our images have some differences in saturation and we have great sensibility on the Saturation channel for our masks we should make each image to be average saturated.

```
def get_mean_saturation_value(files, images_location):
    mean = 0
    n = 0
    for file in files:
        if file[-3:]=='jpg':
            image = cv.imread(images_location+'/'+file)
            image_mean = cv.cvtColor(image,cv.COLOR_BGR2HSV)
            mean = mean + np.mean(image_mean[:,:,1], dtype = 'int32')
            n = n+1

    return mean//n
```

This function gives the average saturation value of our 100 images.

```
new_value = image[i,j,1] - mean + mean_hue

if new_value > 255:
    image_mean_saturation[i,j,1] = 255
elif new_value < 0:
    image_mean_saturation[i,j,1] = 0
else:
    image_mean_saturation[i,j,1] = new_value
```

We will change each pixel of each image so we will have that image with average saturation.

```
mask_red = cv.erode(mask_red, kernel)
mask_red = cv.dilate(mask_red, kernel)

mask_white = cv.erode(mask_white, kernel)
mask_white = cv.dilate(mask_white, kernel)
```

For each mask we will apply erode to remove background noise and dilate after that to keep the same size of the letters.

```
try:
    for i in range(H):
        for j in range(W):
            if mask[i, j] != 0:
                mask[i:i+135, :] =0
                raise Found
except Found:
    pass
```

Here we remove the red rectangle from the top of the board. When we find the first line with a white pixel (in red mask) we will make the next 135 (~the height of the rectangle) lines black. The same idea goes to the rectangle from the bottom left.

```
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
    if possible_top_right is None or point[0] - point[1] > possible_top_right[0] - possible_top_right[1]:
        possible_top_right = point
    if possible_bottom_left is None or -point[0] + point[1] > -possible_bottom_left[0] + possible_bottom_left[1]:
        possible_bottom_left = point
```

Here we apply the idea of finding x, y which have maximum/minimum values for some functions. [0] represents the x axis and [1] the y axis. contours[i] is a component of connected white pixels in the mask. We have to go through all components to find the maximum area i.e. the area of the border.

```
if cv.contourArea(np.array([[possible_top_left],[possible_top_right],[possible_bottom_right],[possible_bottom_left]])) > max_area:
    max_area = cv.contourArea(np.array([[possible_top_left],[possible_top_right],[possible_bottom_right],[possible_bottom_left]]))
    top_left = possible_top_left
    bottom_right = possible_bottom_right
    top_right = possible_top_right
    bottom_left = possible_bottom_left
```

Now we will use the built-in function from open-cv to wrap the image and cut the board

```
def crop_board(top_left,top_right,bottom_right,bottom_left, width, height, image):

    puzzle = np.array([top_left,top_right,bottom_right,bottom_left], dtype = "float32")
    destination_of_puzzle = np.array([[0,0],[width,0],[width,height],[0,height]], dtype = "float32")

    M = cv.getPerspectiveTransform(puzzle, destination_of_puzzle)

    image_rgb = cv.cvtColor(image, cv.COLOR_HSV2BGR)
    board = cv.warpPerspective(image_rgb, M, (width, height), flags = cv.BORDER_REFLECT + cv.WARP_FILL_OUTLIERS)

    return board
```

We are cutting the letters with 20 pixels more on horizontal and 12 on vertical, then apply a mask for the letters.

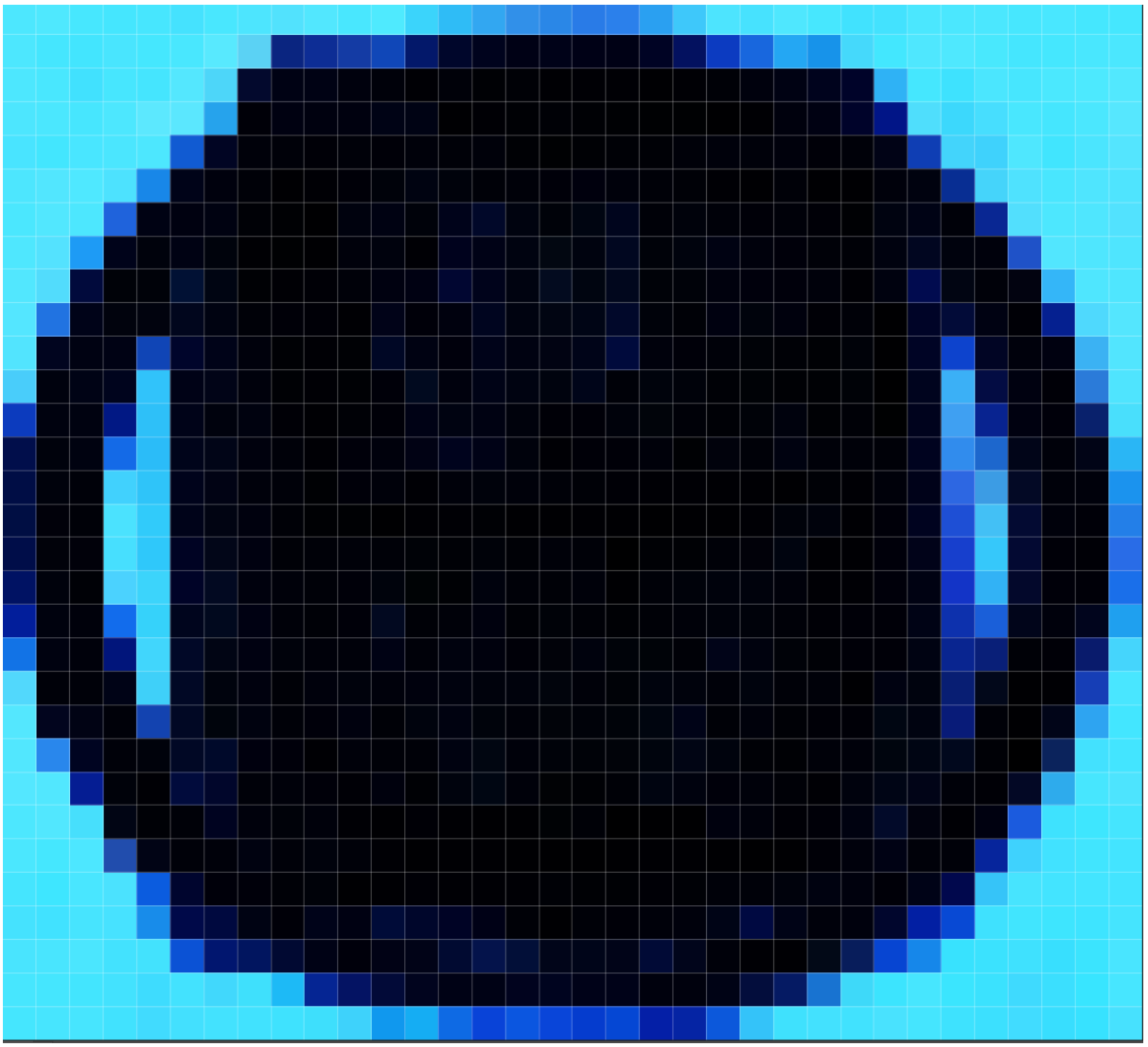
```
patch = cut_patch(board, lines_horizontal, lines_vertical, i, j, 20, 12)
patch = cv.resize(patch, (0,0), fx=2, fy=2, interpolation=cv.INTER_NEAREST)
patch[:,:,2] = cv.medianBlur(patch[:,:,2], 3)

patch = cv.cvtColor(patch, cv.COLOR_BGR2HSV)

lower = np.array([100, 0, 0], np.uint8)
upper = np.array([130, 255, 80], np.uint8)
mask = cv.inRange(patch.copy(), lower, upper)

if np.mean(mask) > 0:
    matrix[i][j] = '+'
else:
    matrix[i][j] = '-'
```

Task 2 - classifying letters

Solving idea

Here we will use the concept of template matching, for this we will need masks for each letter. Those masks were made by saving the warped board (from task 1) and after that taking that image into photoshop. There we used “crop warp tool” to cut each letter, after that we used “spot healing brush tool” and “clone stamp tool” where needed to remove the numbers and the small circles. We aligned all the images to fit in the smallest square possible. We also applied some contrast to the mask.

All masks overlaid using “Darken” layer mode



We want the mask to be as small as possible because some letters get out of their border and so, the mask may miss a good overlay.

How does the code work?

We will use both the path and the mask in gray so we will not have problems with color balance changes.

```
gray_patch = cv.cvtColor(patch, cv.COLOR_BGR2GRAY)

maxi=-np.inf
letter=-1

for j in range(1, 24):

    img_template=cv.imread(mask_location+'/'+str(j)+'.jpg')
    img_template=cv.cvtColor(img_template, cv.COLOR_BGR2GRAY)

    corr = cv.matchTemplate(gray_patch, img_template, cv.TM_CCOEFF_NORMED)
    corr=np.max(corr)
    if corr>maxi:
        maxi=corr
        letter=letters[j]

return letter
```

“corr” first represents an image, the pixel with the highest intensity in that image represents how much it matches with the mask. We will choose the mask that matches the most.

```
matrix[i][j] = classify_letter(mask_location, patch, k)
```

We will store this classification in a matrix.

Task 3 - score calculation

Solving idea

Here the idea is to keep a matrix that contains only the new letters (added in the current round of the game) and one matrix with all other letters. Doing so we will be able to find if a new letter is in a special position or not and it will be easy to find all the new words formed.

How does the code work?

This function starts from the newly added letters stored in the “diff” matrix and checks if there is any new word in each direction of the letters.

```
def get_all_words_created(diff, matrix):
    all_words = diff.copy()

    for i in range(diff.shape[0]):
        for j in range(diff.shape[1]):

            if diff[i][j] != '-':

                k = 1
                while i-k >= 0:
                    if matrix[i-k][j] != '-':
                        all_words[i-k][j] = matrix[i-k][j]
                    else:
                        break
                    k = k + 1
                k = 1
                while i+k < 15:
                    if matrix[i+k][j] != '-':
                        all_words[i+k][j] = matrix[i+k][j]
                    else:
                        break
                    k = k + 1
                k = 1
                while j-k >= 0:
                    if matrix[i][j-k] != '-':
                        all_words[i][j-k] = matrix[i][j-k]
                    else:
                        break
                    k = k + 1
                k = 1
                while j+k < 15:
                    if matrix[i][j+k] != '-':
                        all_words[i][j+k] = matrix[i][j+k]
                    else:
                        break
                    k = k + 1
                k = 1

    return all_words
```

Now that we have the matrix of new words formed we can go through it and calculate the points. We will use the “diff” matrix to find if we have to use the special spots or not. Depending on the direction that we are looking we may find only one letter in that direction, but this is not a word according to Scrabble rules, so we should not at points for this case. The below code is for words written from right to left, we will to the same for word written from top to bottom (by inverting i an j in the two for’s).

```
for i in range(all_words.shape[0]):
    word_points = 0
    double = False
    triple = False
    len = 0
    is_new_word = False
    for j in range(all_words.shape[1]):
        if all_words[i][j] != '-':
            len = len + 1
            if diff[i][j] != '-':
                is_new_word = True
                if special_spot[i][j] == 0:
                    word_points = word_points + letter_points[all_words[i][j]]
                elif special_spot[i][j] == 2:
                    word_points = word_points + 2*letter_points[all_words[i][j]]
                elif special_spot[i][j] == 3:
                    word_points = word_points + 3*letter_points[all_words[i][j]]
                elif special_spot[i][j] == 4:
                    word_points = word_points + letter_points[all_words[i][j]]
                    double = True
                elif special_spot[i][j] == 6:
                    word_points = word_points + letter_points[all_words[i][j]]
                    triple = True
            else:
                word_points = word_points + letter_points[all_words[i][j]]

    if double:
        word_points = 2*word_points
    if triple:
        word_points = 3*word_points

    if len > 1 and is_new_word:
        points = points + word_points
```

We can also check easy how many letters were used by passing “diff” matrix to this function.

```
def check_all_letters_used(matrix):
    count = 0
    for i in range(matrix.shape[0]):
        for j in range(matrix.shape[1]):
            if matrix[i][j] != '-':
                count = count + 1
    if count == 7:
        return 50
    else:
        return 0
```