Gherasim Rareș
Grupa 343

Project – Sudoku and Cell extraction

- **Problem:** You get a path to a file which contains multiple images taken by phone of a sudoku game on a paper. You should also add the number of images to the script.
  - **task_1** assumes that all images contain normal sudoku games. After running the script for **task_1**, it should be able to extract information from the image about the sudoku, like: which cells are occupied, which are not and for bonus it should also be able to recognize numbers and where they appear in the sudoku.
  - **task_2** assumes that all images contain jigsaw sudoku games. After running the script for **task_2**, it should be able to extract information from the image about the sudoku, like: which cells are occupied, which are not, how each sudoku is separate in different zones and, for bonus, it should also be able to recognize numbers and where they appear in the sudoku.

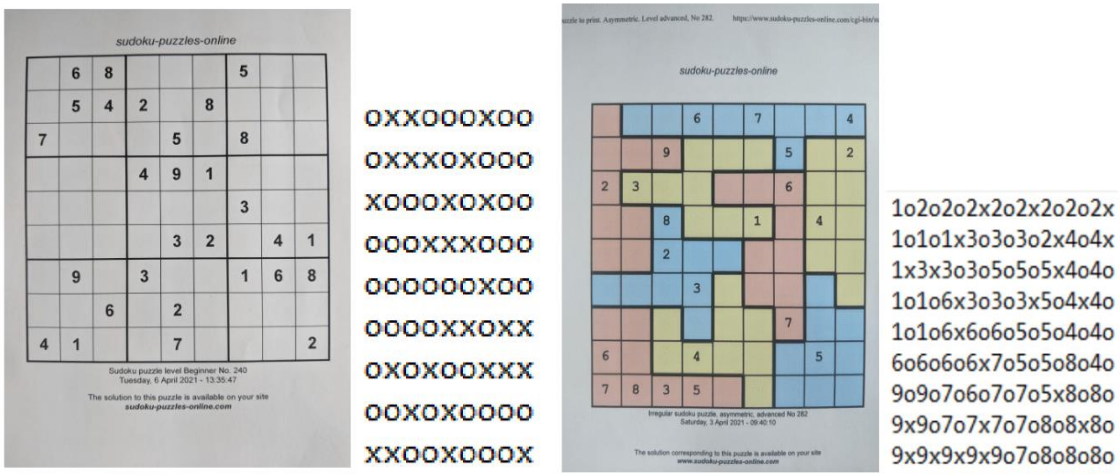


Fig. 1. Normal sudoku game and the resulted configuration (without numbers) and jigsaw sudoku game and the resulted configuration (with numbers)

- **Read images:**
  - Each task should receive, in order to run, the following:
    - The number of images that should be processed.
    - The path to the file where the images are stored.

  Ex:

```
number_of_images = 40
path_to_images = './Tema1/antrenare/jigsaw/'
```

- **Preprocessing for both types (task_1 and task_2) of images:**
  - Using the list of images read from the file, we apply to each of them the following:
    - We convert each of them to gray:
      - *cv2.cvtColor(aux_image, cv2.COLOR_BGR2GRAY)*
    - And we apply an „otsu” threshold:
      - *cv2.threshold(aux_image, 10, 255,cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)*

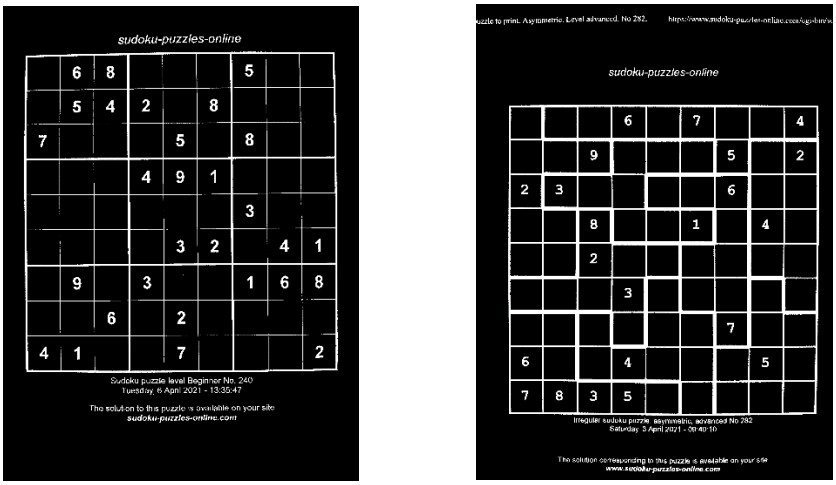


Fig. 2. Left– Preprocessed normal sudoku image; Right– Preprocessed jigsaw sudoku image.

- **Rotates image upright for both types (task_1 and task_2):**
  - We need to rotate each image upright. We do so by first calculating where the corners of the biggest square (sudoku) in the image are.

```
squares, _ = cv2.findContours(image, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
biggest_pos = 0
biggest_square = squares[0]
for i in range(len(squares)):
    if cv2.contourArea(squares[i]) > biggest_pos:
        biggest_square = squares[i]
        biggest_pos = cv2.contourArea(squares[i])

rectang = cv2.minAreaRect(biggest_square)
cornerPoints = cv2.boxPoints(rectang)
cornerPoints = np.int0(cornerPoints)
```

Fig. 3. How we get the biggest square from each image

  - While the differences of the y-axis of the upper-left corner and the upper-right corner is bigger then 20, we totate the image by 0.8 and recalculate the sudoku corners. We stop when their y-axis difference is lower then 20.

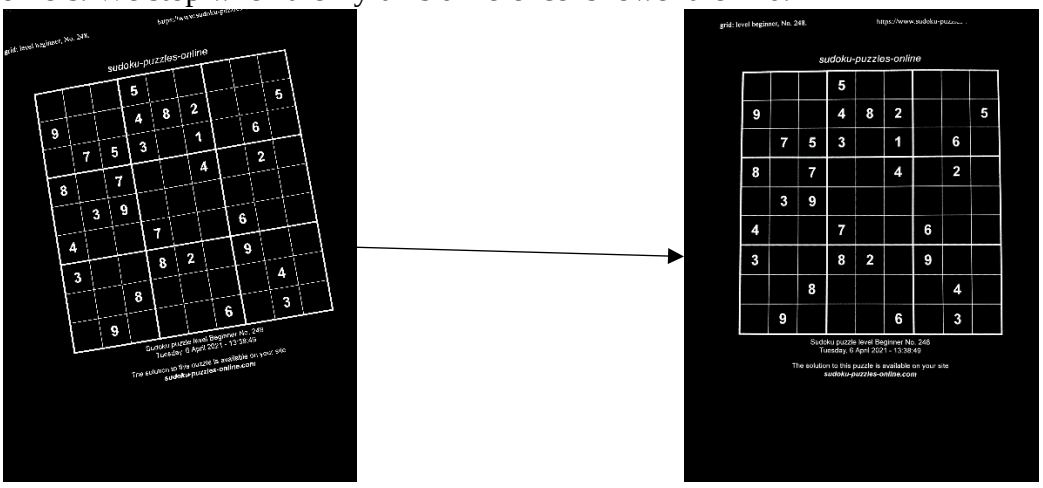


Fig. 4. Left – Image before rotation applied; Right – Image

Gherasim Rareș
Grupa 343

- **Crop the sudoku area from the image for both types (task_1 and task_2):**
  - Using the same corners extracted previously, we cut the sudoku are out from the rest of the image.

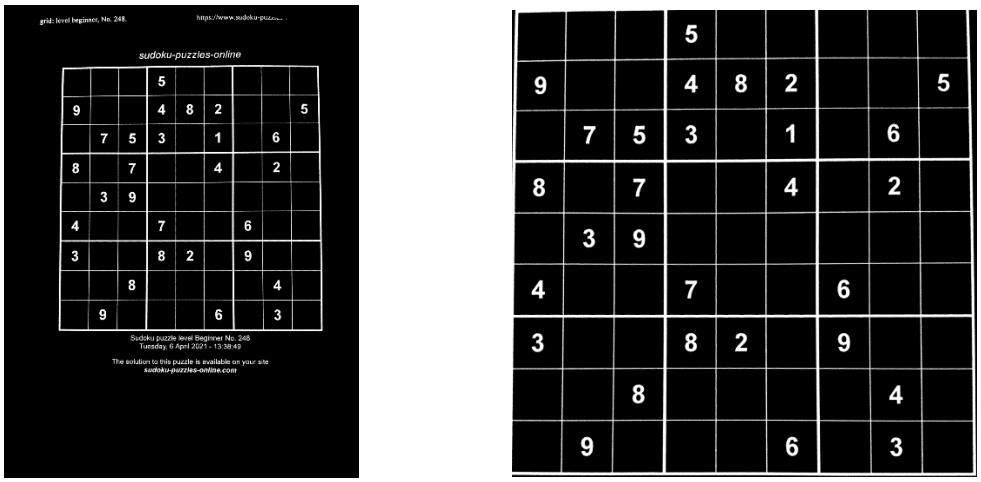


Fig. 5. Left – Prelucrated image; Right – Only the sudoku extracted from the image.

Only task_1:

- **Break each image in 81 tiles**, each tile representing a small square from the big sudoku. We do so by using the width and height of each image and, base on them, predicts where each square is.

```
breaked_images = list()
    for image in images:
        image_tiles = list()
        width, height = image.shape[1] , image.shape[0]
        for line in range(9):
            curr_line = list()
            for col in range(9):
                curr_line.append(image[height//9 * line + offset :
height//9 * (line+1) - offset, width//9 * col + offset: width//9 * (col+1)
- offset])
            image_tiles.append(curr_line)
        breaked_images.append(image_tiles)
```

- **Generate the structure matrix.** In order to generate it, we get all tiles from an images (the breaked ones) and we count how many white pixels each tile has inside of it. If it has more then 500 we consider it a tile with a number inside of it, and without otherwise. If the tile contains a number we give it an „x”, otherwise an „o”.

- **Structure matrix is the result of task_1 so we write it to the result file.** The last part of the task_1 is script (besides the bonus part) is the writting of the results in a file. Each file will contain the coresponding predictions of an image.

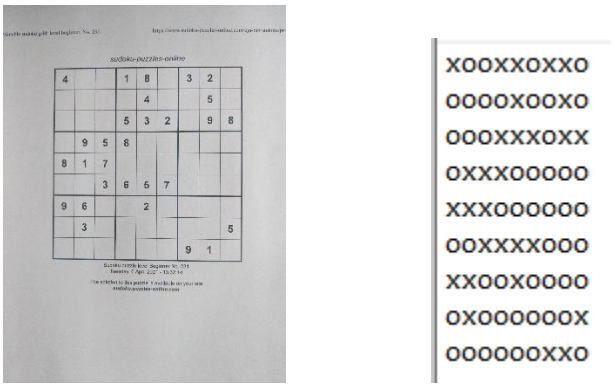


Fig. 6. Left – Original Image; Right – the resulted configuration.

Only task 1 bonus:

- **Before the run:** Inside the file, we shall have a training_data file, which contains multiple .png images (tile images). Also, there should be another file with their classification (what number is inside of each tile) in a training_data.txt.

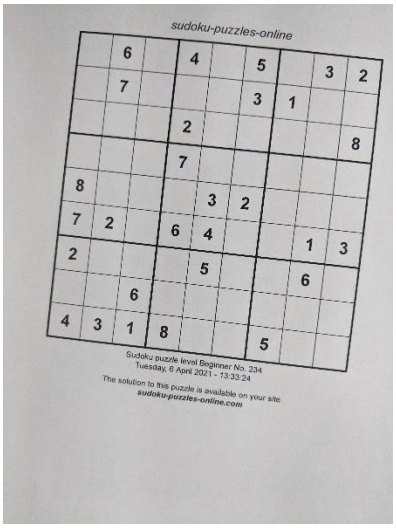
- **Initializing and training of classifier:** We use previous mentioned data to train a svm.SVC classifier, with ‚rbf' kernel and C=4. This classifier should be able to recognize what number is inside of each tile, based on its training.
    - From tests, I managed to obtain 91% accuracy, separating the images in training and validation data, training ones being 80% of total and validation 20%. Now all images are set as training ones.

```
        self.classifier = svm.SVC(kernel = 'rbf', C=4)
```

- **Predict each tile number:** When generating the result, we use the structure_matrix predifined.
    - If the structure matrix contains an ‚x', we try to predict that tile number using the classifier. Otherwise, we live it like that (when it contains ‚o').

```
prediction = clasiffier.predict(breaked_sudoku_images[i][line//3][collumn])
```

- **We write the results in a file.**




```
o6o4o5o32
o7ooo31oo
ooo2oooo8
ooo7ooooo
8ooo32ooo
72o64oo13
2ooo5oo6o
oo6oooooo
4318oo5oo
```

Fig. 7. Left – Original Image; Right – Final result after predictions.

Only task 2:

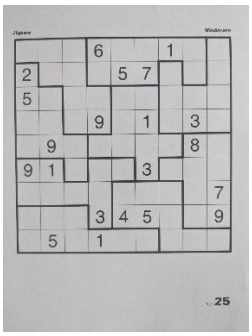
- **Detect strong lines (the ones which separate two different zone):**
    - The only difference between a classic sudoku and a jigsaw ones are the zones, which can be easily detect if you can detect strong lines which separate the zones.
    - Detecting **vertical strong lines** and **horizontal strong lines** is basically the same.
        - We use the width and height of the cropped sudoku image to assume where each line is going to be.
        - We extract all lines and count how many white pixels each of them has and add this number to a list.
        - We calculate the mean of the sum of all the numbers in the list.
        - We go again though each line, count the number of white pixels again and:
            - If it has more then the mean, it means that the line is a strong one and we mark it with a ‚1’.
            - If it has less then the mean, it means that the line is a normal one and we mark it with a ‚0’.

Gherasim Rareș
Grupa 343

- **Create zone matrix:**
  - Zone matrix will contain to which zone is each tile is part of.
  - Initially all zones in the matrix are 0. (not allocated)
  - We go though the matrix usign a recursive approach, adding 1 to the first position and trying to go left-right-up-down and add 1 again. We can go to a certain direction only if it exists in the matrix, is equal to 0 and there is not strong line between the current position and it.

```
def recursive_mark_squares(zone_matrix, curr_number, index_line, index_col, vertical_strong_lines, horizontal_strong_lines):
    zone_matrix[index_line][index_col] = curr_number

    if index_line != 0:
        if zone_matrix[index_line - 1][index_col] == 0 and horizontal_strong_lines[index_line - 1][index_col] == 0:
            recursive_mark_squares(zone_matrix, curr_number, index_line-1, index_col, vertical_strong_lines, horizontal_strong_lines)

    if index_line != 8:
        if zone_matrix[index_line + 1][index_col] == 0 and horizontal_strong_lines[index_line][index_col] == 0:
            recursive_mark_squares(zone_matrix, curr_number, index_line + 1, index_col, vertical_strong_lines, horizontal_strong_lines)

    if index_col != 0:
        if zone_matrix[index_line][index_col - 1] == 0 and vertical_strong_lines[index_line][index_col - 1] == 0:
            recursive_mark_squares(zone_matrix, curr_number, index_line, index_col - 1, vertical_strong_lines, horizontal_strong_lines)

    if index_col != 8:
        if zone_matrix[index_line][index_col + 1] == 0 and vertical_strong_lines[index_line][index_col ] == 0:
            recursive_mark_squares(zone_matrix, curr_number, index_line, index_col + 1, vertical_strong_lines, horizontal_strong_lines)

def create_zone_matrix(vertical_strong_lines, horizontal_strong_lines):
    zone_matrix = [
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
    ]
    curr_number = 1
    for index_line in range(9):
        for index_col in range(9):
            if zone_matrix[index_line][index_col] == 0:
                recursive_mark_squares(zone_matrix, curr_number, index_line, index_col, vertical_strong_lines, horizontal_strong_lines)
                curr_number += 1
    return zone_matrix
```

- **We join the structure matrix and the zone matrix.**
  - Now that we have the structure matrix and the zone matrix, we join them so that we know if a square is empty or not and if it has a number inside of it or not.
- **We write the results in a file.**



```
1o1o1o2x2o2o2x2o3o
4x1o1o2o2x2x3o2o3o
4x4o1o1o5o5o3o3o3o
4o4o1o1x5o5x3o3x3o
4o4x4o5o5o5o5o6x6o
7x7x4o7o7o5x6o6o6o
7o7o7o7o8o8o8o6o6x
9o9o9o7x8x8x8o6o6x
9o9x9o9x9o9o8o8o8o
```

Fig. 8. Left – Original Image; Right – The result.

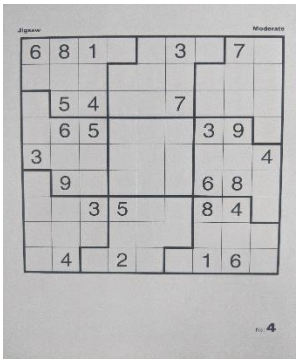
Only task 2 bonus:

- **Before the run:** Inside the file, we shall have a training_data_jigsaw file, which contains multiple .png images (tile images). Also, there should be another file with their classification (what number is inside of each tile) in a training_data_jigsaw.txt.

- **Initializing and training of classifier:** We use previous mentioned data to train a svm.SVC classifier, with ‚rbf' kernel and C=4. This classifier should be able to recognize what number is inside of each tile, based on its training.
  - From tests, I managed to obtain 95% accuracy, separating the images in training and validation data, training ones being 80% of total and validation 20%. Now all images are set as training ones.

```
        self.classifier = svm.SVC(kernel = 'rbf', C=4)
```

- **Create predictions:** We go though the joined result of structure matrix and zone matrix and for each ‚x' that we have, we try to run our classifier against it and generate a prediction of the number that is inside of that tile. The prediction then replaces the ‚x' in the joined_matrix and we print the result in a file.



```
1618111o2o232o373o
1o1o1o2o2o2o3o3o3o
4o15142o2o273o3o3o
4o46455o5o5o63693o
434o4o5o5o5o6o6o64
7o494o5o5o5o66686o
7o7o73858o8o98946o
7o7o7o8o8o8o9o9o9o
7o748o828o9o91969o
```

Fig. 9. Left – Original Image; Right – The result with numbers predicted.