Introduction

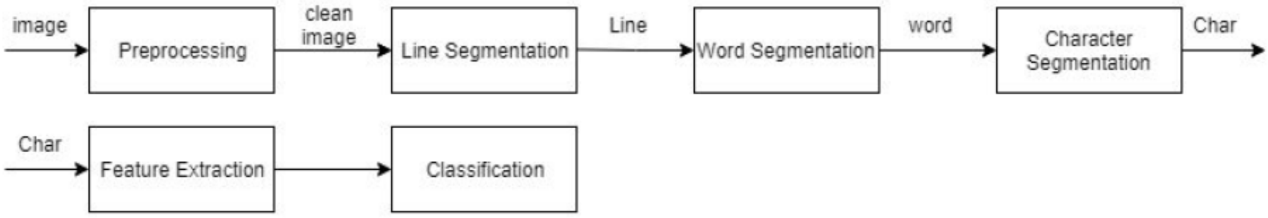
Arabic-OCR is a Python program written by HusseinYoussef on GitHub that is basically an OCR for Arabic text. For some reason, I am unable to build it properly on my computer (considering that it was updated 10 months ago) so this small little report is a study on how the code works so that we can possibly learn off of it in order to create the Jawi OCR.

The Program

The program is run through the command line, running

```
python OCR.py
```

in which it returns an output folder with the text results in a folder for each image. HusseinYoussef also provides a simple pipeline for the program:



Let's go through each step of the program.

Running the program

When you run the program, it creates an output directory for the files, and notably runs a function called `run()` on the image path provided. `run()` is where a good bulk of the code lives.

```
def run(image_path):
    # Read test image
    full_image = cv.imread(image_path)
    predicted_text = ''

    # Start Timer
    before = time.time()
    words = extract_words(full_image)       # [ (word, its line),(word,
its line),..  ]
...
```

Here we see that it reads the image path to provide the image, and starts a timer as well. Then it will run a function called `extract_words()` that lives in `segmentation.py`. (There's more to this function, but later...)

```
def extract_words(img, visual=0):

    lines = line_horizontal_projection(img)
    words = []

    for idx, line in enumerate(lines):

        if visual:
            save_image(line, 'lines', f'line{idx}')

        line_words = word_vertical_projection(line)
        for w in line_words:
            words.append((w, line))

        if visual:
        for idx, word in enumerate(words):
            save_image(word[0], 'words', f'word{idx}')

    return words
```

This function returns all the words in the image, assumably as little images. We will go into detail about these functions below...

Preprocessing

I assume this takes part in the function present in `segmentation.py`, specifically called `preprocess`.

```
def preprocess(image):

    # Maybe we end up using only gray level image.
    gray_img = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    gray_img = cv.bitwise_not(gray_img)

    binary_img = binary_otsus(gray_img, 0)
        deskewed_img = deskew(binary_img)

        return deskewed_img
```

The function converts the image to a gray image using OpenCV, and then performs a `bitwise_not` to invert the colors. Then it binarizes the image using `binary_otsus` and finally deskews it. Notably, this function is called within the `line_horizontal_projection` function, presumably as part of line segmentation.

Line + Word segmentation

Both line and word segmentation use the same helper function, `projection_segmentation()`, so I'll talk about it in the same header.

```
def projection_segmentation(clean_img, axis, cut=3):

    segments = []
    start = -1
    cnt = 0

    projection_bins = projection(clean_img, axis)
    for idx, projection_bin in enumerate(projection_bins):

        if projection_bin != 0:
            cnt = 0
        if projection_bin != 0 and start == -1:
            start = idx
        if projection_bin == 0 and start != -1:
            cnt += 1
            if cnt >= cut:
                if axis == 'horizontal':
                    segments.append(clean_img[max(start-1, 0):idx, :])
                elif axis == 'vertical':
                    segments.append(clean_img[:, max(start-1, 0):idx])
                cnt = 0
                start = -1

    return segments
```

It creates `projection_bins` by calling the helper function `projection()` which computes the sum of pixel intensities along either the rows or columns of the image.

Then it loops over the projection bins with this if statement:

- **Non-zero Projection Bin**
  - It will reset the `cnt` to 0 (indicating end of a zero bin streak)
  - If `start == 1` then it will set it to the index, indicating a start of a segment.
- **Zero Projection Bin**
  - It will increment `cnt` indicating we are in the start of a segment.
  - If `cnt >= cut` (indicating a sufficient number of 0s on the projection, default is 3), then the segment is 'finished', and it will slice the segment out of the image.
  - The segment will be appended to the `segments` list, and then it resets everything and restarts.

For line segmentation, the program calls `projection_segmentation` on a horizontal axis, and passes a list of lines back. Similarly, for word segmentation, the program then calls `projection_segmentation` on a vertical axis on each line provided, now passing a list of words back.

This then leads to character segmentation.

Character segmentation

Going back to `run()`, we can carry on to see more of the function.

```
    pool = mp.Pool(mp.cpu_count())
    predicted_words = pool.map(run2, words)
    pool.close()
    pool.join()
    # Stop Timer
    after = time.time()

    # append in the total string.
    for word in predicted_words:
        predicted_text += word
        predicted_text += ' '

    exc_time = after-before
    # Create file with the same name of the image
    img_name = image_path.split('\\')[1].split('.')[0]

    with open(f'output/text/{img_name}.txt', 'w', encoding='utf8') as fo:
        fo.writelines(predicted_text)

    return (img_name, exc_time)
```

Now we can see there is a `pool.map(run2, words)` section. This maps `run2()` over all the words we got from `extract_words()`.

```
def run2(obj):
    word, line = obj
    model = load_model()
    # For each word in the image
    char_imgs = segment(line, word)
    txt_word = ''
    # For each character in the word
    for char_img in char_imgs:
        try:
            ready_char = prepare_char(char_img)
        except:
            # breakpoint()
            continue
```

```
            feature_vector = featurizer(ready_char)
            predicted_char = model.predict([feature_vector])[0]
            txt_word += predicted_char
    return txt_word
```

This is where the OCR comes into play -- the word images are being passed through a pre-trained model in order to identify what word it is. We'll talk about training the model later on.

Then, it calls `segment()` to perform character segmentation on the words provided.

```
def segment(line, word_img):

    binary_word = word_img//255
    no_dots_copy = remove_dots(binary_word)
...
```

First, we binarize the word image, and then we remove the dots on the word.

```
def remove_dots(word_img, threshold=11):

    no_dots = word_img.copy()

    components, labels, stats, GoCs = 
cv.connectedComponentsWithStats(no_dots, connectivity=8)
    char = []
    for label in range(1, components):
        _, _, _, _, size = stats[label]
        if size > threshold:
            char.append(label)
    for label in range(1, components):
        _, _, _, _, size = stats[label]
        if label not in  char:
            no_dots[labels == label] = 0

    return no_dots
```

First, we make a copy of the image, and then do connected components analysis, which performs connected component labeling on the image:

- `components`: The number of connected components found.
- `labels`: An image where each pixel is labeled with the component number it belongs to.
- `stats`: An array containing statistics of each component (e.g., bounding box, area).
- `GoCs`: The centroid of each component.

Then, they go through all the labels, and find the size of each component. If the size of the label is bigger than the threshold (default is 11) then it is considered a character and not a dot.

It then loops again through the sizes, and if the label is not considered a character, it is removed, and finally returns the image.

Going back to `segment()`:

```
        ...
        VP_no_dots = projection(no_dots_copy, 'vertical')
    VP = projection(binary_word, 'vertical')
    binary_word = fill(binary_word, VP_no_dots)
    no_dots_copy = remove_dots(binary_word)
    ...
```

Next, the function calculates the vertical projection of the word with no dots, and the projection of the word with dots. Then, it calls the `fill()` function, and seems to reset `binary_word` and `no_dots_copy` to its original form. **I'm not 100% sure why it does this, but I'll try and figure it out.**

Moving back to `segment()`:

```
...
    upper_base, lower_base, MFV = baseline_detection(remove_dots(line))
    MTI = horizontal_transitions(no_dots_copy, upper_base)
...
```

The function then does baseline detection on the line with its dots removed, to determine the upper and lower baselines of the projection, as well as its thickness.

Following that, it performs `horizontal_transitions()` using the `no_dots_copy` of the image as well as the upper baseline of the line.

```
def horizontal_transitions(word_img, baseline_idx):

    max_transitions = 0
    max_transitions_idx = baseline_idx
    line_idx = baseline_idx-1
    lines = []
    # new temp image with no dots above baseline

    while line_idx >= 0:
        current_transitions = 0
        flag = 0

        horizontal_line = word_img[line_idx, :]
```

```
        for pixel in reversed(horizontal_line):

            if pixel == 1 and flag == 0:
                current_transitions += 1
                flag = 1
            elif pixel == 0 and flag == 1:
                current_transitions += 1
                flag = 0

        if current_transitions >= max_transitions:
            max_transitions = current_transitions
            lines.append(line_idx)
            max_transitions_idx = line_idx

        line_idx -= 1

    return lines[len(lines)//2]
```

It seems that this function is analyzing all the horizontal lines in the image, to try and determine which line has the maximum number of transitions from 0 to 1 -- and then returns the median of those with the maximum transactions.

It starts from the baseline and analyzes each horizontal line above it...**again, not sure why we need this, but good to know.**

Going back to `segment()`, we have:

```
...
    SRL, wrong = cut_points(binary_word, VP, MFV, MTI, upper_base)

    if wrong:
        MTI -= 1
        SRL.clear()
        SRL, wrong = cut_points(binary_word, VP, MFV, MTI, upper_base)

    HP = projection(line, 'horizontal')
    top_line = -1

    valid = filter_regions(binary_word, no_dots_copy, SRL, VP, upper_base, 
lower_base, MTI, MFV, top_line)

    chars = extract_char(binary_word, valid)

    return chars
```

We pass `cut_points()` the binarized word, the vertical projection of the word (`VP`), the thickness of the baseline (`MFV`), the median of the transition (`MTI`). This is a very long function, so we'll give it it's own section here, so we will discuss it later on.

However, if we have anything that is wrong, we lower the `MTI`, clear the `SRL`, and try it again.

Finally, we get the horizontal projection of the line `HP` and set the `top_line` to `-1`.

Next, it calls `filter_regions()`. This is another painful and long code block, so we will analyze it separately here, but it returns the valid regions of separation in the word.

Finally, we can call `extract_char()` on the word using the `valid` regions of separation previously calculated.

```
def extract_char(img, valid_SR):

    # binary image needs to be (0, 255) to be saved on disk not (0, 1)
    img = img * 255
    h, w = img.shape

    next_cut = w
    char_imgs = []

    for SR in valid_SR:
        char_imgs.append(img[:, SR[1]:next_cut])
        next_cut = SR[1]
    char_imgs.append(img[:, 0:next_cut])

    return char_imgs
```

In this function, the image is just segmented and cut up into various shapes judging by the characters discovered in it.

`segment()` then returns these characters.

Feature extraction

The character images are then prepared using a function called `prepare_char()`.

```
def prepare_char(char_img):

    binary_char = binarize(char_img)

    try:
        char_box = bound_box(binary_char)
        resized = cv.resize(char_box, dim, interpolation = cv.INTER_AREA)
    except:
        pass

    return resized
```

It binarizes the character image, and then attempts to calculate the bound box of the character using `bound_box()`. After finding the bound box, it resizes the image to fit it better, and returns it.

Then, it's passed to the `featurizer()` function, where it is...just flattened, apparently.

Finally, the model `predicts` what picture it is, and adds it to the string `txt_word`, and finally returns the word!

But how did we get the model?

Training

While training, HusseinYoussef tries 4 different classifiers: `LinearSVC()`, `MLPClassifier()` twice, once with 1 layer, and again with 2 layers, and `GaussianNB()`, or Gaussian Naive Bayes.

In the end, it produced these levels of accuracy:

```
Score of LinearSVM: 0.9891379310344828
Score of 1L_NN: 0.9952586206896552
Score of 2L_NN: 0.9968103448275862
Score of Gaussian_Naive_Bayes: 0.8507327586206896
```

Judging by this, it is a good idea to use either `LinearSVM` or `MLPClassifier`, but the only issue of using `MLPClassifier` is that it requires a lot of computational resources to run properly, so perhaps if I am training it on my own PC, then I should try using LinearSVM instead.

Below is his code for training.

```
def train():

    X, Y = read_data()
    assert(len(X) == len(Y))

    X, Y = shuffle(X, Y)

    X_train = []
    Y_train = []
    X_test = []
    Y_test = []

    X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
train_size=0.8)

    X_train = np.array(X_train)
```

```
    Y_train = np.array(Y_train)
    X_test = np.array(X_test)
    Y_test = np.array(Y_test)
...
```

In this part of the code, he separates the data into training and testing sets, and then makes them into arrays for better training.

```
    scores = []
    for idx, clf in tqdm(enumerate(classifiers), desc='Classifiers'):

        if not skip[idx]:

            clf.fit(X_train, Y_train)
            score = clf.score(X_test, Y_test)
            scores.append(score)
            print(score)

            # Save the model
            destination = f'models'
            if not os.path.exists(destination):
                os.makedirs(destination)

            location = f'models/{names[idx]}.sav'
            pickle.dump(clf, open(location, 'wb'))


    with open('models/report.txt', 'w') as fo:
        for score, name in zip(scores, names):
            fo.writelines(f'Score of {name}: {score}\n')
```

Then for each classifier, he fits the classifier to the training set, creates a testing score and prints it into a file called `report.txt`.

Conclusion

In conclusion, it seems that in order to have a good training, we need to have a lot of samples of different characters in the Jawi script, as well as needing to be familiar with all the types of strokes and dots that make up the script in order to describe the dots and strokes and other things like that. HusseinYoussef has three articles that he uses as references, so I will study those three articles as well to determine more information.

Papers Referenced by HusseinYoussef

A. Qaroush, B. Jaber, K. Mohammad, M. Washha et al., "# An Efficient, Font Independent Word and Character Segmentation Algorithm for Printed Arabic Text," *Journal of King Saud University - Computer and Information Sciences*., vol 34, no. 1, pp. 1-15, 2019. [Online].

Available: https://www.researchgate.net/publication/335562626_An_Efficient_Font_Independent_Word_and_Character_Segmentation_Algorithm_for_Printed_Arabic_Text [Accessed May. 17, 2024]

M. Ayesh, K. Mohammad, A. Qaroush, S. Agaian et al., "A Robust Line Segmentation Algorithm for Arabic Printed Text with Diacritics", *Electronic Imaging*., vol 2017, no. 13, pp. 42-47, 2017. [Online]. Available: https://www.researchgate.net/publication/317876029_A_Robust_Line_Segmentation_Algorithm_for_Arabic_Printed_Text_with_Diacritics [Accessed May. 17, 2024]

M. A. A. Mousa, M. S. Sayed, M. I. Abdalla. (July 2017) Arabic Character Segmentation Using Projection Based Approach with Profile's Amplitude Filter. Presented at the ICoICT 2013: International Conference of Information and Communication Technology. [Online]. Available: https://www.researchgate.net/publication/318205989_Arabic_Character_Segmentation_Using_Projection_Based_Approach_with_Profile's_Amplitude_Filter

---

Appendix

Cut_Points

Let's go through the function `cut_points()`.

```
def cut_points(word_img, VP, MFV, MTI, baseline_idx):

    # flag to know the start of the word
    f = 0

    flag = 0
    (h, w) = word_img.shape
    i = w-1
    separation_regions = []

    wrong = 0
...
```

First, it initializes the flag for the start of the word, gets the shape of the image, and initializes both a wrong list and a list of separation regions.

```
    # loop over the width of the image from right to left
    while i >= 0:

        pixel = word_img[MTI, i]

```

```
        if pixel == 1 and f == 0:
            f = 1
            flag = 1
...
```

It will loop through the image from left to right, and get the pixel. If the pixel is white (`pixel==1`), then the word is started -- setting `f` and `flag` to 1.

```

        if f == 1:

            # Get start and end of separation region (both are black
pixels <----)
            if pixel == 0 and flag == 1:
                start = i+1
                flag = 0
            elif pixel == 1 and flag == 0:
                end = i          # end maybe = i not i+1
                flag = 1
                mid = (start + end) // 2
                    ...
```

If a black pixel (`pixel==0`) is found while `flag` is 1, then it's marked as the start of a separation region. If a white pixel is found while `flag` is 0, then the separation region ends, and the midpoint is calculated.

```
                left_zero = -1
                left_MFV = -1
                right_zero = -1
                right_MFV = -1
                # threshold for MFV
                T = 1

                j = mid - 1
                # loop from mid to end to get nearest VP = 0 and VP = MFV

                while j >= end:

                    if VP[j] == 0 and left_zero == -1:
                        left_zero = j
                    if VP[j] <= MFV + T and left_MFV == -1:
                        left_MFV = j

                    # if left_zero != -1 and left_MFV != -1:
                    #     break

                    j -= 1
```

```
                j = mid
                # loop from mid to start to get nearest VP = 0 and VP =
MFV
                while j <= start:

                    if VP[j] == 0 and right_zero == -1:
                        right_zero = j
                    if VP[j] <= MFV + T and right_MFV == -1:
                        right_MFV = j

                    if right_zero != -1 and right_MFV != -1:
                        break

                    j += 1

                # Check for VP = 0 first
                if VP[mid] == 0:
                    cut_index = mid
                elif left_zero != -1 and right_zero != -1:

                    if abs(left_zero-mid) <= abs(right_zero-mid):
                        cut_index = left_zero
                    else:
                        cut_index = right_zero
                elif left_zero != -1:
                    cut_index = left_zero
                elif right_zero != -1:
                    cut_index = right_zero

                # Check for VP = MFV second
                # elif VP[mid] <= MFV+T:
                #     cut_index = mid
                elif left_MFV != -1:
                    cut_index = left_MFV
                elif right_MFV != -1:
                    cut_index = right_MFV
                else:
                    cut_index = mid
                                ...
```

This code looks for the closest columns to the midpoint of where the vertical projection `VP[j] == 0`, or where it is close to the most frequent value (`VP[j]) <= MFV + T` where `T` is a threshold, and then decides the cut index based on this.

```
                seg = word_img[:, end:start]
                HP = projection(seg, 'horizontal')
                SHPA = np.sum(HP[:MTI])
```

```
                SHPB = np.sum(HP[MTI+1:])

                top = 0
                for idx, proj in enumerate(HP):
                    if proj != 0:
                        top = idx
                        break

                cnt = 0
                for k in range(end, cut_index+1):
                    if vertical_transitions(word_img, k) > 2:
                        cnt = 1
                if SHPB == 0 and (baseline_idx - top) <= 5 and cnt == 1:
                    # breakpoint()
                    wrong = 1
                else:
                    separation_regions.append((end, cut_index, start))

        i -= 1

    return separation_regions, wrong
```

Next, get the horizontal projection of the segment we cut, and then check over it. We mark it as wrong if it meets some certain circumstances. Finally, we return `separation_regions` and `wrong`.

Filter_Regions

Below is the code for `filter_regions()`.

```
def filter_regions(word_img, no_dots_copy, SRL:list, VP:list,
upper_base:int, lower_base:int, MTI:int, MFV:int, top_line:int):

    valid_separation_regions = []
    overlap = []

    T = 1
    components, labels= cv.connectedComponents(word_img[:lower_base+5, :],
connectivity=8)
...
```

`filter_regions()` takes a lot of options -- the `word_img`, a copy of it with no dots, assuming `SRL` is probably a list of separation regions, `VP` - the vertical projection profile, `upper_base` and `lower_base`, along with the median text-line index `MTI`, the most frequent value `MFV`, and the `top_line` as well.

```
...
    SR_idx = 0
    while SR_idx < len(SRL):

        SR = SRL[SR_idx]
        end_idx, cut_idx, start_idx = SR
...
```

Now it will loop through each separation region in `SRL`, and extracts the `end_idx`, `cut_idx`, and `start_idx` from the separation region. Then, we go through the many cases that might consider it a valid separation point.

```
...
                # Case 1 : Vertical Projection = 0
        if VP[cut_idx] == 0:
            valid_separation_regions.append(SR)
            SR_idx += 1
            continue
...
```

Case 1: Vertical Projection = 0
If there is no vertical projection, then it is a valid separation region.

```
...
      # Case 2 : no connected path between start and end
        if labels[MTI, end_idx] != labels[MTI, start_idx]:
            valid_separation_regions.append(SR)
            overlap.append(SR)
            SR_idx += 1
            continue
...
```

Case 2: No connected path between start and end
If there is no connected path between start and end (a gap?) then it is a valid separation region.

```
...
        # Case 3 : Contain Holes
        cc, l = cv.connectedComponents(1-(no_dots_copy[:,
end_idx:start_idx+1]), connectivity=4)

        if cc-1 >= 3 and inside_hole(no_dots_copy, end_idx, start_idx):
            SR_idx += 1
            continue
...
```

Case 3: Contains holes
It gets the connected components, and then runs a custom function called `inside_hole()` to check if the segment has a hole or not. If it has a hole, then we skip over it.

```
...
        # Case 4 : No baseline between start and end
        segment = no_dots_copy[:, end_idx+1: start_idx]
        segment_width = start_idx-end_idx-1

        j = end_idx+1
        cnt = 0
        while j < start_idx:

            # Black pixel (Discontinuity)
            base = upper_base-T
            while base <= lower_base+T:

                pixel = no_dots_copy[base][j]
                cnt += pixel

                base += 1

            j += 1

        if cnt < segment_width-2 and segment_width > 4:

            segment_HP = projection(segment, 'horizontal')

            SHPA = np.sum(segment_HP[:upper_base])
            SHPB = np.sum(segment_HP[lower_base+T+1:])

            if (int(SHPB) - int(SHPA)) >= 0:
                SR_idx += 1
                continue
            elif VP[cut_idx] <= MFV + T:
                valid_separation_regions.append(SR)
                SR_idx += 1
                continue
            else:
                SR_idx += 1
                continue
...
```

Case 4: No Baseline Between Start and End
Checks that there are no significant baseline presence between the start and end indices by summing the horizontal projections above and below the baseline.

```
...
        # Case 5 : Last region or next VP[nextcut] = 0
        if SR_idx == len(SRL) - 1 or VP[SRL[SR_idx+1][1]] == 0:

            if SR_idx == len(SRL) - 1:
                segment_dots = word_img[:, :SRL[SR_idx][1]+1]
                segment = no_dots_copy[:, :SRL[SR_idx][1]+1]
                next_cut = 0
            else:
                next_cut = SRL[SR_idx+1][1]
                segment_dots = word_img[:, next_cut:SRL[SR_idx][1]+1]
                segment = no_dots_copy[:, next_cut:SRL[SR_idx][1]+1]

            segment_HP = projection(segment, 'horizontal')
            (h, w) = segment.shape

            top = -1
            for i, proj in enumerate(segment_HP):
                if proj != 0:
                    top = i
                    break
            height = upper_base - top

                        SHPA = np.sum(segment_HP[:upper_base])
            SHPB = np.sum(segment_HP[lower_base+T+1:])
            sk = skeletonize(segment).astype(np.uint8)
            seg_VP = projection(segment, 'vertical')
            non_zero =  np.nonzero(seg_VP)[0]
            cnt = 0

            for k in range(0, 3):
                if k >= len(non_zero):
                    break
                index = non_zero[k]
                if seg_VP[index] >= height:
                    cnt += 1

            if (SHPB <= 5 and cnt > 0 and height <= 6) or (len(non_zero)
>= 10 and SHPB > SHPA and not check_dots(segment_dots)):
                SR_idx += 1
                continue
...
```

Case 5: Last Region or Next VP[nextcut] = 0

If we are in the last region, or the vertical projection `VP` of the next cut is 0, then it will use projections and skeletonization to assess whether it is a valid separation point, based on the present of text above/below the baseline, and whether there are dots.

```
...
        # Strokes

        SEGP = (-1, -1)
        SEG = (-1, -1)
        SEGN = (-1, -1)
        SEGNN = (-1, -1)
        SEGP_SR1 = (0, 0)
        SEGP_SR2 = (0, 0)
        SEG_SR1 = (0, 0)
        SEG_SR2 = (0, 0)
        SEGN_SR1 = (0, 0)
        SEGN_SR2 = (0, 0)
        SEGNN_SR1 = (0, 0)
        SEGNN_SR2 = (0, 0)

        current_cut = SR[1]
        ...
```

First, we start with the initialization of all the different strokes.

```
        ...
        if SR_idx == 0:
            SEGP = (SRL[SR_idx][1], word_img.shape[1]-1)
            SEGP_SR1 = (SRL[SR_idx][0], SRL[SR_idx][2])
            SEGP_SR2 = (SRL[SR_idx][1], word_img.shape[1]-1)

        if SR_idx > 0:
            SEGP = (SRL[SR_idx][1], SRL[SR_idx-1][1])
            SEGP_SR1 = (SRL[SR_idx][0], SRL[SR_idx][2])
            SEGP_SR2 = (SRL[SR_idx-1][0], SRL[SR_idx-1][2])

        if SR_idx < len(SRL)-1:
            SEG = (SRL[SR_idx+1][1], SRL[SR_idx][1])
            SEG_SR1 = (SRL[SR_idx][0], SRL[SR_idx][2])
            SEG_SR2 = (SRL[SR_idx+1][0], SRL[SR_idx+1][2])

        if SR_idx < len(SRL)-2:
            SEGN = (SRL[SR_idx+2][1], SRL[SR_idx+1][1])
            SEGN_SR1 = (SRL[SR_idx+1][0], SRL[SR_idx+1][2])
            SEGN_SR2 = (SRL[SR_idx+2][0], SRL[SR_idx+2][2])
        elif SR_idx == len(SRL)-2:
            SEGN = (0, SRL[SR_idx+1][1])
            SEGN_SR1 = (SRL[SR_idx+1][0], SRL[SR_idx+1][2])
            SEGN_SR2 = (0, SRL[SR_idx+1][2])


        if SR_idx < len(SRL)-3:
```

```
            SEGNN = (SRL[SR_idx+3][1], SRL[SR_idx+2][1])
            SEGNN_SR1 = (SRL[SR_idx+2][0], SRL[SR_idx+2][2])
            SEGNN_SR2 = (SRL[SR_idx+3][0], SRL[SR_idx+3][2])
...
```

Depending on the different indexes, we determine the neighbouring regions.

```
        # SEG is stroke with dots
        if SEG[0] != -1 and\
            (check_stroke(no_dots_copy, no_dots_copy[:, SEG[0]:SEG[1]],
upper_base, lower_base, SEG_SR1, SEG_SR2) \
            and check_dots(word_img[:, SEG[0]:SEG[1]])):

            # Case when starts with ش
            if SEGP[0] != -1 and \
                ((check_stroke(no_dots_copy, no_dots_copy[:,
SEGP[0]:SEGP[1]], upper_base, lower_base, SEGP_SR1, SEGP_SR2) \
                and not check_dots(word_img[:, SEGP[0]:SEGP[1]]))\
                and (SR_idx == 0 or VP[SRL[SR_idx-1][1]] == 0 or
(VP[SRL[SR_idx-1][1]] == 0 and SRL[SR_idx-1] in overlap))):

                SR_idx += 2
                continue
            else:
                valid_separation_regions.append(SR)
                SR_idx += 1
                continue

        # SEG is stroke without dots
        elif SEG[0] != -1\
            and (check_stroke(no_dots_copy, no_dots_copy[:,
SEG[0]:SEG[1]], upper_base, lower_base, SEG_SR1, SEG_SR2) \
            and not check_dots(word_img[:, SEG[0]:SEG[1]])):

            # Case starts with س
            if SEGP[0] != -1\
                and (check_stroke(no_dots_copy, no_dots_copy[:,
SEGP[0]:SEGP[1]], upper_base, lower_base, SEGP_SR1, SEGP_SR2) \
                and not check_dots(word_img[:, SEGP[0]:SEGP[1]])):

                SR_idx += 2
                continue

            # SEGN is stroke without dots
            if SEGN[0] != -1 \
                and (check_stroke(no_dots_copy, no_dots_copy[:,
SEGN[0]:SEGN[1]], upper_base, lower_base, SEGN_SR1, SEGN_SR2) \
                and not check_dots(word_img[:, SEGN[0]:SEGN[1]])):
```

```
                valid_separation_regions.append(SR)
                SR_idx += 3
                continue

            # SEGN stroke with Dots and SEGNN stroke without Dots
            if SEGN[0] != -1\
                and (check_stroke(no_dots_copy, no_dots_copy[:,
SEGN[0]:SEGN[1]], upper_base, lower_base, SEGN_SR1, SEGN_SR2) \
                and check_dots(word_img[:, SEGN[0]:SEGN[1]])) \
                and ((SEGNN[0] != -1 \
                and (check_stroke(no_dots_copy, no_dots_copy[:,
SEGNN[0]:SEGNN[1]], upper_base, lower_base, SEGNN_SR1, SEGNN_SR2) \
                and not check_dots(word_img[:, SEGNN[0]:SEGNN[1]]))) or
(len(SRL)-1-SR_idx == 2) or (len(SRL)-1-SR_idx == 3)):

                    valid_separation_regions.append(SR)
                    SR_idx += 3
                    continue

            # SEGN is not stroke or Stroke with Dots
            if SEGN[0] != -1 \
                and ((not check_stroke(no_dots_copy, no_dots_copy[:,
SEGN[0]:SEGN[1]], upper_base, lower_base, SEGN_SR1, SEGN_SR2)) \
                or (check_stroke(no_dots_copy, no_dots_copy[:,
SEGN[0]:SEGN[1]], upper_base, lower_base, SEGN_SR1, SEGN_SR2) \
                and check_dots(word_img[:, SEGN[0]:SEGN[1]]))):

                    SR_idx += 1
                    continue

            SR_idx += 1
            continue
...
```

Then it goes through a bunch of different conditions: checking for strokes with dots, for strokes without dots, cases when they start with a certain icon. Based on some of these conditions, then it is appended to the separation regions.

```
...
        if (len(valid_separation_regions) == 0 or\
            len(valid_separation_regions) > 0 and abs(cut_idx-
valid_separation_regions[-1][1]) > 2):
            valid_separation_regions.append(SR)
        SR_idx += 1

    return valid_separation_regions
```

Then it checks whether there are any empty regions, then the current separation region is added to the valid_separation_region without anything else. If there *are* other regions, then it checks the distance between the last region, and if it's more than 2, then we add it.

Finally, it returns the `valid_separation_regions`.