Bubble sheet evaluator

Introduction:

Bubble sheet evaluator is an OCR (mini) project where the ultimate goal is to automate the bubble answer sheet evaluating process. The main objective is to analyze the given scanned bubble answer sheets and perform crossed examination to the correct answers to evaluate the obtained marks.

Objectives:

- Build cmd tools to automate bubble sheets evaluation process
- Integrate it into a mobile app to perform real-time bubble sheet evaluation.

Module used:

- openCV
- Numpy
- Imutils
- Matplotlib
- Statistics
- pandas

Overview of Proposed Approaches:

1. **Approach I: clustering**

   Cluster is a number of things of the same kind, growing or held together. Every item of a cluster is similar to each other and should be dissimilar with every item of other clusters.

   Hypothesis:

   - Based on area, perimeter, center of contour, it is assumed that contours belonging to bubbles (circle/rectangle) shape on the sheet should be grouped together to form a cluster. On the other hand, contours other than of bubbles' should belong to another cluster.

2. **Approach II: Histogram**

   A histogram is the most commonly used graph to show frequency distributions. In this approach we will use histogram to analyze the distribution of the area of contours (could be of bubbles or noises).

   Hypothesis:

   - It is assumed that every bubble has a similar area and the frequency of the area of contours of those bubbles will be the highest among the area of contours of noises in a sheet.

Phase I: Contour detection

Flow chart:
Preprocessing stage will be common for both approaches.

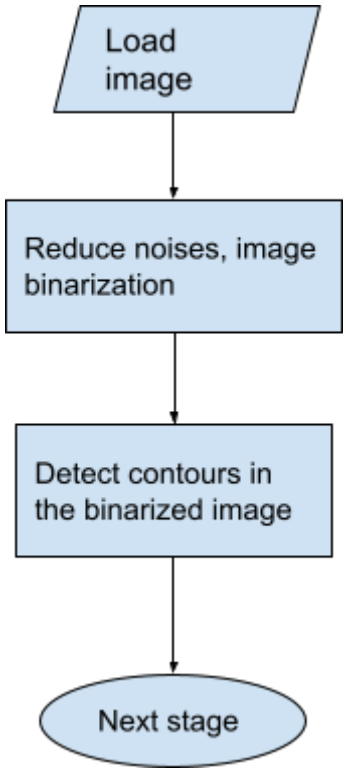


Fig: flow chart of phase I (contour detection)

Approach I: Clustering

- **Phase II: Contour filtering**

In this phase, we try to filter out noises from the raw contour list which is detected by the openCV function.

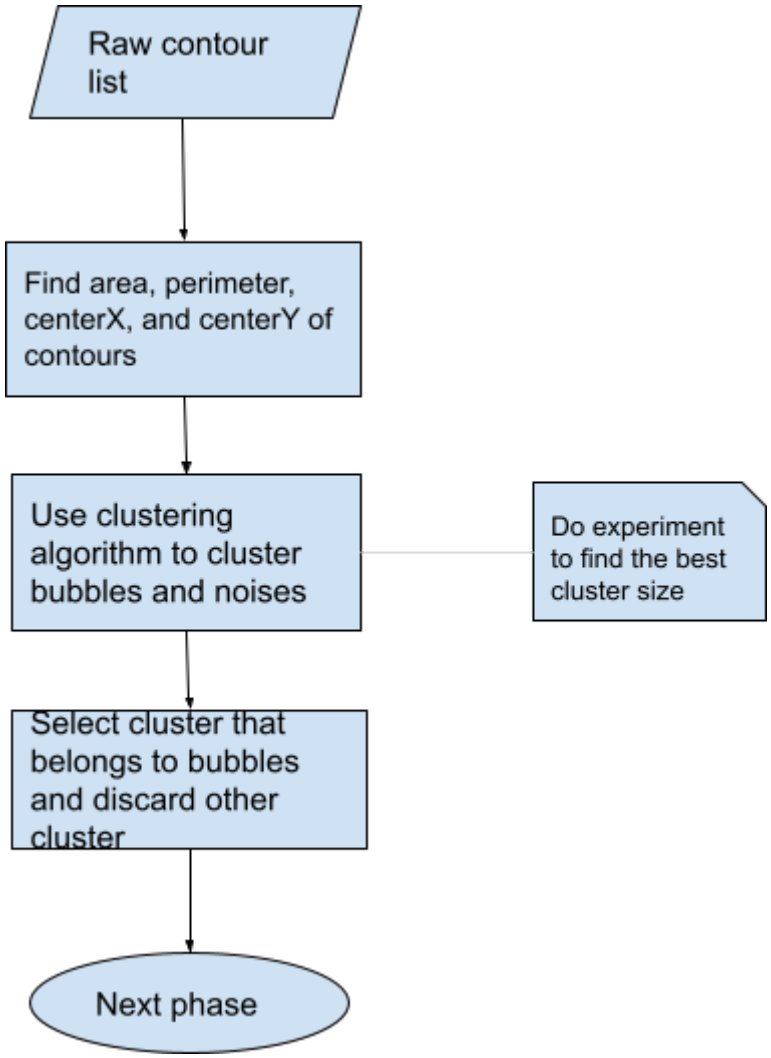


Fig: flow chart of phase II of approach I

Result of cluster analysis:

- **AgglomerativeClustering**

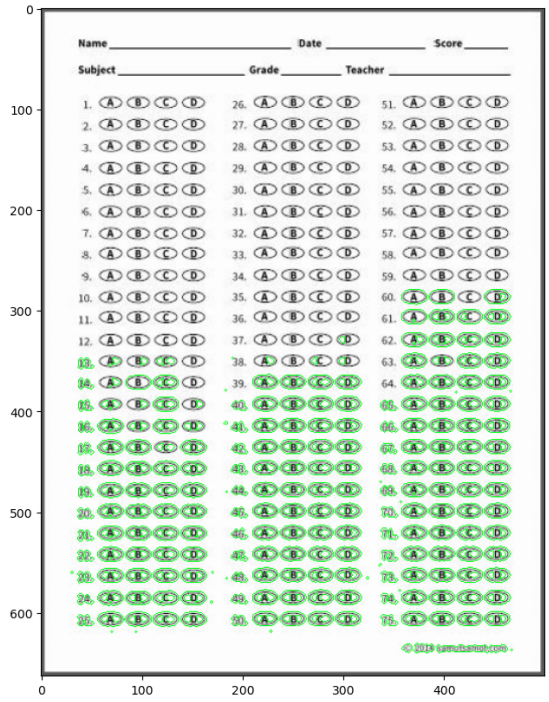


Fig: bubbles of one cluster after performing cluster analysis are highlighted and plotted into the corresponding bubble answer sheet

- **KMeans Clustering**

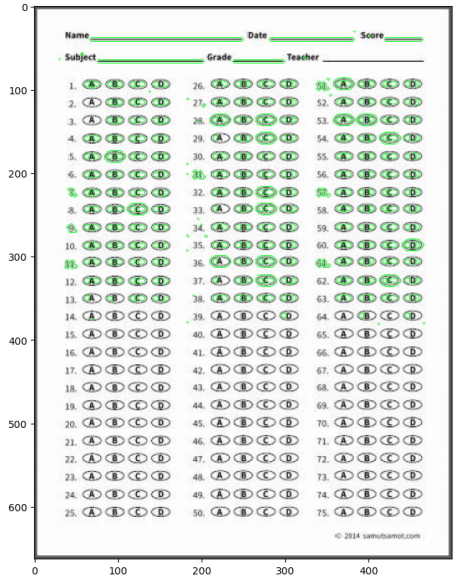


Fig: bubbles of one cluster after performing cluster analysis are highlighted and plotted into the corresponding bubble answer sheet

- **Cluster size**

```
In [313]: for i in range(0, n_clusters):
              cluster_bucket.append(contour_feature[contour_feature["label"] == i])
              print(f"cluster {i} shape: ",len(cluster_bucket[i]))

          cluster 0 shape:  549
          cluster 1 shape:  152
          cluster 2 shape:  373
          cluster 3 shape:  1
```

Fig: size of 4 clusters after performing cluster analysis

Conclusion:

Upon performing experiments on various sheets, and featuring engineering, it is found that cluster analysis does not give the perfect result as assumed. The two main problems of this approach which makes it unacceptable are:

- Only the bubbles of a particular area are clustered together.
- Only by looking at the cluster size, it is difficult to distinguish the cluster of noises and bubbles.

Approach II: Histogram

- **Phase II: Contour filtering**
    - In this phase we try to filter out contours of noises based on histogram of contours area.

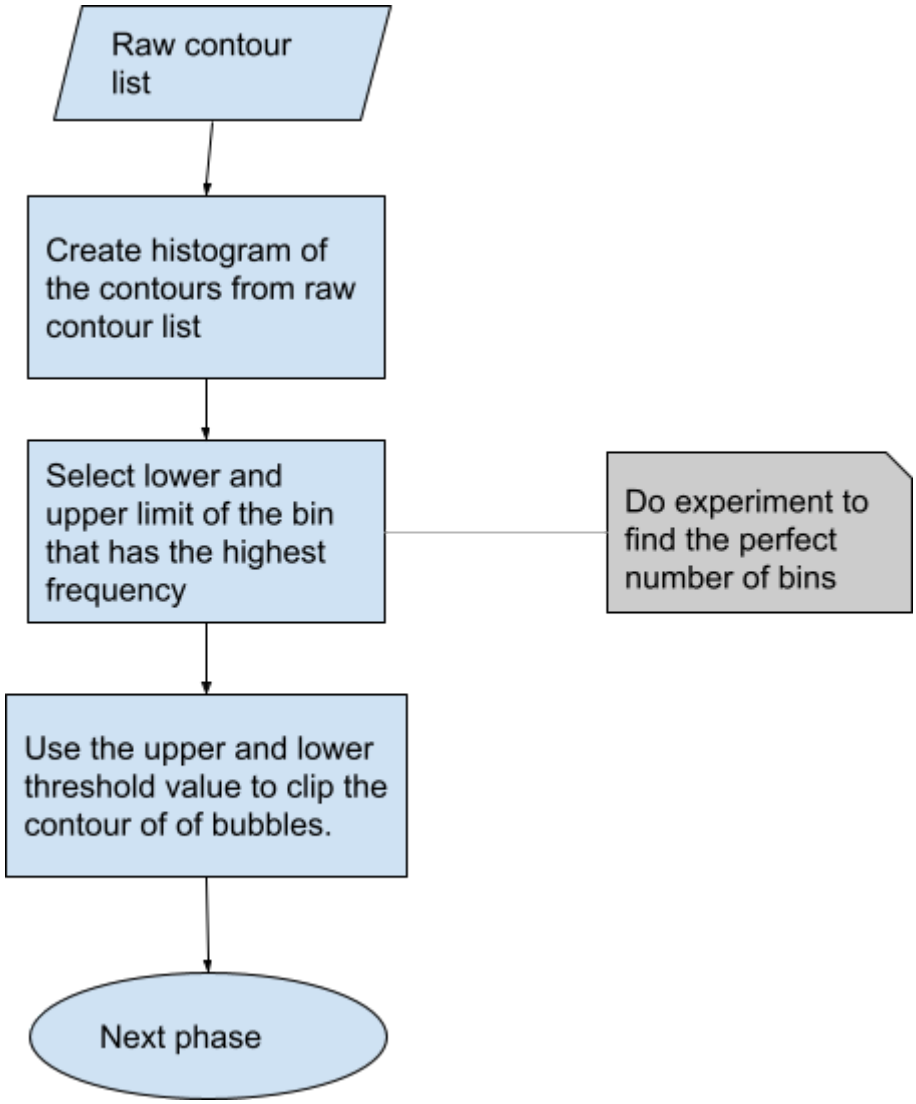


Fig: flow chart of phase II of approach II

Findings After Experiment:

After experimenting by doing actual coding, it was found that we can discard the contours that have area > 500 and area < 100, they belong to noise.

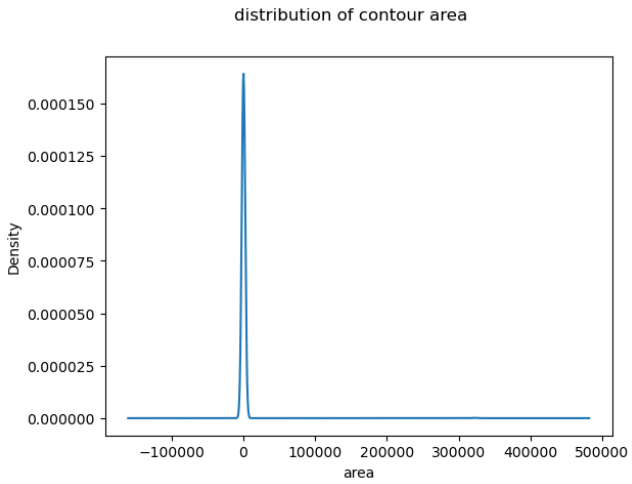


Fig: distribution of contour areas without performing any clipping of sample 1

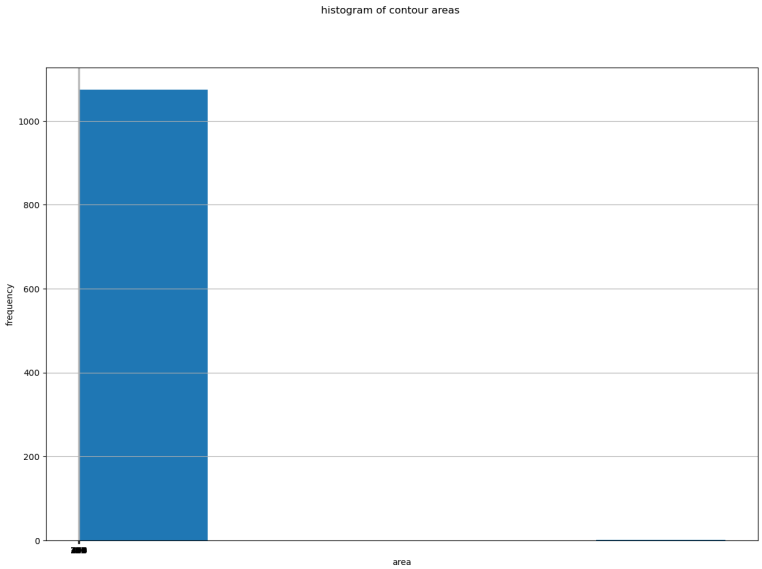


Fig: histogram of contour area without performing any clipping of sample 1

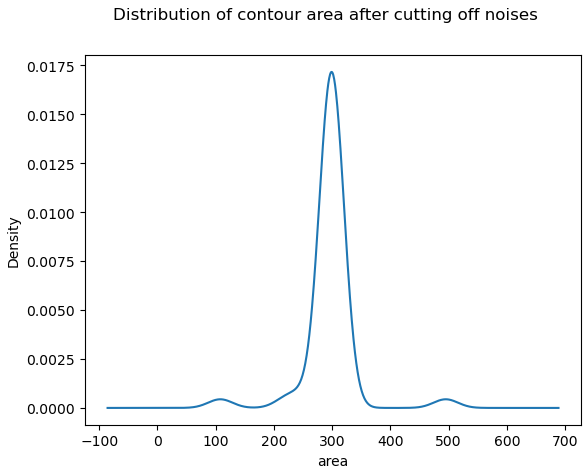


Fig: distribution of contour area where 100 < area < 500

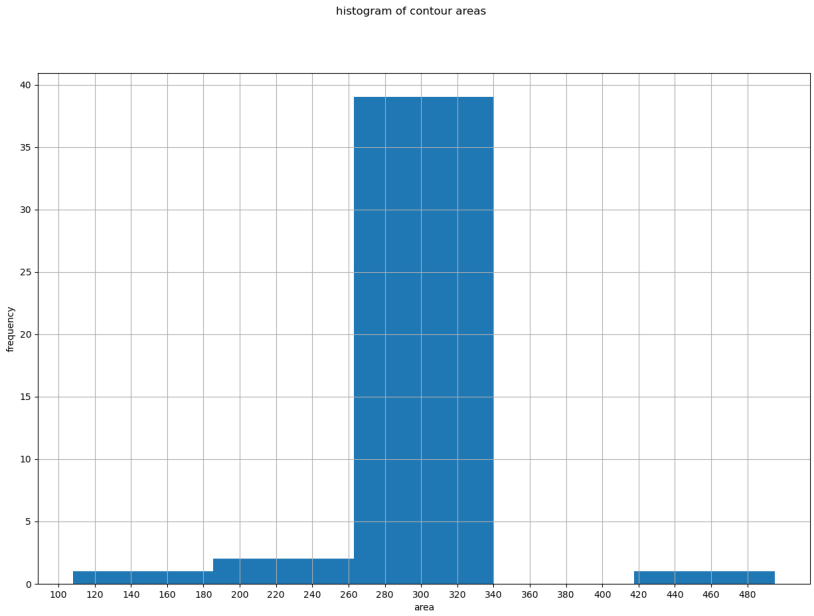


Fig: histogram of contour area where 100 < area < 500

Challenge I

- **How to Automatically Detecting the lower and upper threshold value of the highest bin?**

  We have to create a custom function that creates a dict of histogram.

```python
def generate_histogram(list_, n_bins=4):
    max_value, min_value = max(list_), min(list_)
    bin_size = (max_value - min_value)/n_bins
    bins_ranges = [(min_value+i*bin_size, min_value+(i+1)*bin_size) for i in range(0, n_bins)]
    histogram = {x:0 for x in range(len(bins_ranges))}
    for i in list_:
        for index, bin_ in enumerate(bins_ranges):
            if i < bin_[1] and  i >= bin_[0]:
                histogram[index] += 1
    return histogram, bins_ranges
```

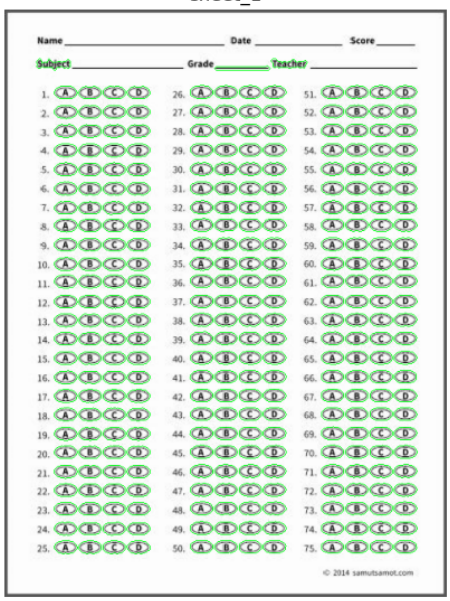
Fig: function that generates histogram

```python
def get_max_bin_thresholds(histogram, bins_ranges):
    hist = dict(sorted(histogram.items(), key=lambda item: item[1], reverse=True))
    return bins_ranges[list(hist.keys())[0]]
```

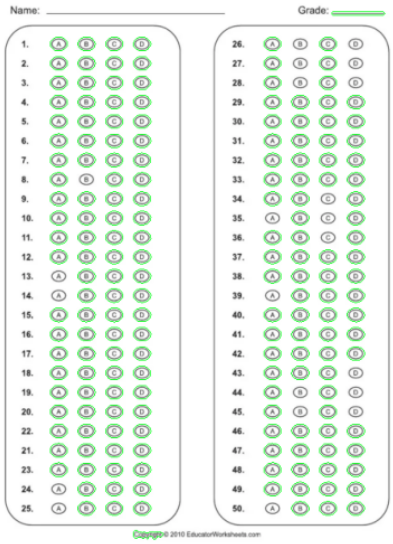
Fig: function that returns the min threshold and max threshold of the highest bin

Result After Histogram analysis:

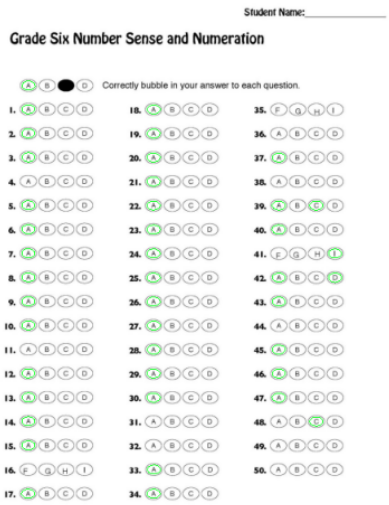
sheet_1



sheet_2



sheet_3



sheet_4

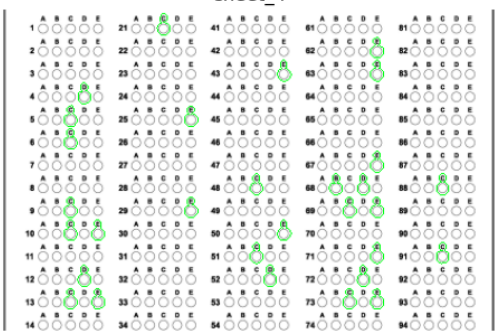


Fig: testing on different sample answer sheet

Conclusion:

We can clearly see that this approach is far better than the previous clustering approach. However, upon analyzing the above result we can not say that this approach is always perfect. We may improve the accuracy by **improving the preprocessing stage** and we also can remove noise in coming phases. Let's assume that we may be able to remove all

noises but we still need to deal with the missing bubble contours. So, in order to tolerate missing bubbles we can implement a robust algorithm for which not every bubble contour is necessary.

- **Phase III:**
  - **Subphase I:**
    grouping contour center of bubbles based on question number

    Here we are going to create a 3D matrix to represent groups of bubbles. We are going to group bubbles that belong to the same question. This 3D matrix is going to capture the geometric structure of the bubbles clusters of the sheet.
    For example, this 3D matrix is going to look like this:

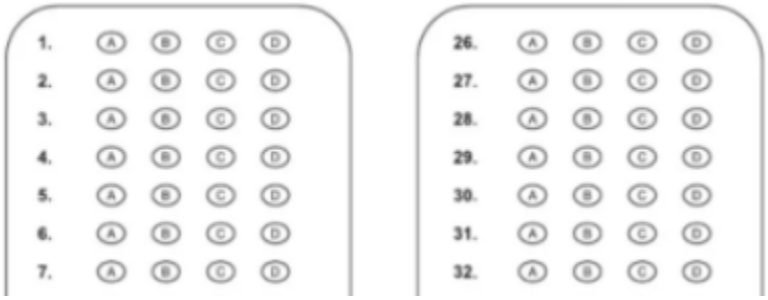


Fig: bubble sheet sample

Corresponding matrix representation of first two rows

```
3D_matrix = [
              [ [(1, 1), (10, 1), (20, 1), (30,1)],      [(50, 1), (61, 1), (72, 1), (80, 1)]],
              [[(1, 10), (11, 10), (19, 10), (30, 10)],  [(50, 10), (60, 10), (71, 10), (80, 10)]
            ]
```

Challenge II:

- **Bubbles that belong to the same rows may not have the same center_Y value, there might be slight variation among them. How to account for this? Similar case for center_X.**

To tackle this problem we need to add a tolerance value that helps to match two coordinates. Value_X == value_Y if and only if value_X < (value_Y + tolerance) and value_X > (value_Y - tolerance). In this way we can capture the variation in the equivalent values.

Code snippet:

- **Function to group the raw bubbles_centers into rows**

```
def groupby_axis_y(bubble_centers, tolerance = 3):
    '''
    parameters:
        * bubble_centers : list of bubble centers
        * tolerance: integer value that specifies the range of variation for equivalent values

    Description:
        This function transforms 1D list of contours into 2D where contours are grouped into rows
        (note that each row may contain corresponding row of more that one columns).
        Grouping is based on Y_coordinate of the bubble center.

    returns:
        * 2D list -> grouped bubbles center
        * min_x, min_y -> minimum x and y coordinate (center of top-left bubble)
        * max_x, max_y -> maximum x and y coordinate
    '''
    groups = defaultdict(list)
    mapper = {}
    # initializing mapper
    x,y = bubble_centers[0]
    mapper[y] = (y-tolerance, y+tolerance)
    groups[y].append((x,y))
    min_x = x
    max_x = x
    min_y = y
    max_y = y
    for x,y in bubble_centers[1:]:
        # grouping bubbles
        flag_found = False
        new_y = None
        for k,v in mapper.items():
            new_y = k
            if v[0] < y and v[1] > y:
                groups[k].append((x,new_y))
                flag_found = True
                break
        # if mapper doesnot have y already
        if not flag_found:
            mapper[y] = (y-tolerance, y+tolerance)
            groups[y].append((x,y))
            new_y = y
        # finding min and max x,y
        if x < min_x:
            min_x = x
        if new_y < min_y:
            min_y = new_y
        if x > max_x:
            max_x = x
        if new_y > max_y:
            max_y = new_y
    return list(groups.values()) , [(min_x, min_y), (max_x, max_y)]
```

Fig: Code snippet of groupby_axis_y function

- **Function to group the row wised grouped bubble_centers into columns**

The main limitation of this groupby_axis_x function is at least there should exist one perfect row, else it will be unable to estimate the the position of missing bubbles.

```
def get_avg_x_spacing(groups_y, ebpg):
    '''
    parameters:
        * groups_y: list of row wised grouped bubbles center
        * ebpg (expected bubble per group): integer value that specifies the number of bubbles for a single question

    Details:
       Ebpg helps us to determine the number of bubbles in each column of a row.
       This function will check for the perfect rows where ever bubble is detected and there is no noise.
       If no perfect rows is availabe then halt.
       Else, find the mode of median of spacing of two consecutive bubbles i.e X2 - X1.
    '''
    perfect_rows_x = [] # stores x coordinates of bubbles of perfect rows
    average_x_spacing = []

    columns_x = defaultdict(list) # stores the x coordinate of first bubble of each column group

    for row in groups_y:
        row = sorted(row)
        spacing  = []
        pointer = 0
        if len(row) % ebpg != 0: # if the row is perfect then when we divide it by the number of bubbles per question the remainc
            continue

        perfect_rows_x.append([x for x,y in row])

        for i in range(1, len(row)):
            if (i-1) % ebpg == 0 and (i-1) != 0:
                columns_x[pointer].append(row[i-1][0])
                pointer = (pointer + 1) % ebpg

            spacing.append(abs(row[i][0] - row[i-1][0]))

        # recording average x spacing of this row
        average_x_spacing.append(statistics.median(spacing))
#         average_x_spacing.append(statistics.mode(spacing))

    if len(average_x_spacing) > 0:
        # find mode of perfect_rows column wise
        perfect_row_x = pd.DataFrame(perfect_rows_x).mode(axis=0)
        columns_x_mode = [statistics.mode(X) for X in columns_x.values()]
        return min(average_x_spacing), columns_x_mode, perfect_row_x.values.tolist()[0]
    else:
        return None
```

Fig: Code snippet of get_avg_x_spacing function

```python
def groupby_axis_x(groups_y, min_x, max_x, tolerance=3, ebpg=4): #ebpg(expected bubble per group)=expected number of options/bub
    '''
    parameters:
        * groups_y: list of row wised grouped bubbles center
        * min_x, max_x = minimum and maximum x_coordinate
        * tolerance =  an integer that specifies the range of variation for equivalent values
        * ebpg (expected bubble per group): integer value that specifies the number of bubbles for a single question

    Description:
        It takes list of grouped bubbles (grouped into rows) and segment each rows into multiple columns.
        For instance; [[1,2,3,4,5,6],[7,8,9,10,11,12]] is grouped into columns as

                [[[1,2,3], [4,5,6]],
                 [[7,8,9], [10,11,12]]]
    Returns:
        * groups: 3D list of grouped bubbles
        * perfect_rows_x: xcoordinate of the bubbles of a perfect rows
    '''
    # setting 2D empty bucket
    groups = [[] for i in range(len(groups_y))]
    avg_distance, columns_x, perfect_row_x = get_avg_x_spacing(groups_y, ebpg)
    column_pointer = 0 # initially pointing to the zero th column
    if avg_distance is None: # there is not row that matches the specified ebpg, mean doesnot exist perfect rows (without missin
        return []
    for i, row in enumerate(groups_y):
        row = sorted(row)
        previous_bubble_center_x = 0
        current_bubble_center_x = 0
        # checking if the first bubble of the row is detected or not
        if row[0][0] > (min_x - tolerance) and row[0][0] < (min_x + tolerance):
            groups[i].append([(row[0])])
            current_bubble_center_x = row[0][0]
        else:
            groups[i].append([(min_x, None)])
            current_bubble_center_x = min_x

        # now checkig for the rest of the bubble of the row
        bubble_index = 1
        while current_bubble_center_x < (max_x-tolerance):

            if len(row) > bubble_index and\
                   (abs(row[bubble_index][0] - current_bubble_center_x) < (avg_distance + tolerance) and\
                   abs(row[bubble_index][0] - current_bubble_center_x) > (avg_distance - tolerance)) :

                    # if one options set is covered then add list for new col group
                    groups[i][len(groups[i])-1].append(row[bubble_index])
                    previous_bubble_center_x = current_bubble_center_x
                    current_bubble_center_x = row[bubble_index][0]
                    bubble_index += 1


            # there is a missing bubble (unable to detect bubble)
            else:
                if len(groups[i][len(groups[i]) - 1]) % ebpg == 0:

                    if len(row) > bubble_index and (row[bubble_index][0] < (columns_x[column_pointer] + tolerance)) and\
                    (row[bubble_index][0] > (columns_x[column_pointer] - tolerance)):

                        groups[i].append([(columns_x[column_pointer], row[bubble_index][1])])
                        bubble_index += 1

                    else:
                        groups[i].append([(columns_x[column_pointer], None)])

                    previous_bubble_center_x = current_bubble_center_x
                    current_bubble_center_x = columns_x[column_pointer]
                    column_pointer = (column_pointer + 1) % len(columns_x)

                else:
                    groups[i][len(groups[i]) - 1].append((None, None))
                    # last bubble of the group + average distance
                    previous_bubble_center_x = current_bubble_center_x
                    current_bubble_center_x = current_bubble_center_x + avg_distance
    return groups, perfect_row_x
```

Fig: Code snippet of groupby_axis_x function

- **Function that replace missing values in a groups by estimated possible value**

```python
def estimate_missing_bubbles(groups, perfect_row_x, ebpg=4): #consecutive bubble distance, row distance, coln distance
    '''
    Parameters:
        * groups : 3D list of grouped bubbles obtained from groupby_axis_x function
        * perfect_row_x: x-coordinate of the bubbles of a perfect rows obtained from groupby_axis_x_function

    Description:
        it will replace the missing values in the groups by estimation the possible coordinate value for the missing bubbles

    Returns:
        * groups: 3D list of grouped bubbles where missing values are replaced with the estimated possible values
    '''
    for row_index, row in enumerate(groups):
        flat_row = [y for col in row for x,y in col]
        for col_index, col in enumerate(row):
            # flattern row
            for bubble_index, bubble in enumerate(col):
                # if x coordinate is missing
                if bubble[0] == None:
                    # convert non liner indexing to linear
                    linear_index = col_index * ebpg + bubble_index

                    new_x = perfect_row_x[linear_index]
                    old_y = groups[row_index][col_index][bubble_index][1]
                    groups[row_index][col_index][bubble_index] = (new_x, old_y)

                # if y coordinate is missing
                if bubble[1] == None:
                    # find not none y or bubble[1]
                    unique_y = list(set(flat_row))

                    new_y = unique_y[0] if unique_y[0] else unique_y[len(unique_y) - 1]
                    old_x = groups[row_index][col_index][bubble_index][0]

                    groups[row_index][col_index][bubble_index] = (old_x, new_y)

    return groups
```

Fig: code snippet of estimate_missing_bubbles function

- **Function to plot rectangular box around each bubbles**

```python
def get_x_y_spacing(groups, margin=5): # margin is the gap betwen two consecutive bubbles
    '''
    Parameters:
        * groups: groups obtained from estimate_missing_bubbles function

    Description:
        It will return the half of the distance between two consecutive bubbles (both X axis and Y axis)

    Returns:
        * h_spacing : half of the distance between two consecutive bubbles of a row 1
        * v_spacing : half of the distance between the first bubbles of two consecutive rows
    '''
    x_spacing = abs(groups[0][0][1][0] - groups[0][0][0][0])
    y_spacing = abs(groups[1][0][1][1] - groups[0][0][0][1])
    return int((x_spacing - margin)//2), int((y_spacing - margin)//2)

def plot_bubble_rec_bbox(groups, image, figsize=(10,10)):
    '''
    Parameters:
        * groups: groups obtained from estimate_missing_bubbles function

    Description:
        It will plot the rectangular bounding box around each bubbles

    Returns:
        None but plots image with bounding rectangular box
    '''
    rect_w, rect_h = get_x_y_spacing(groups[:2])

    for rows in groups:
        for col in rows:
            for bubble in col:
                start_point = int(bubble[0] - rect_w), int(bubble[1] - rect_h)
                end_point = int(bubble[0] + rect_w), int(bubble[1] + rect_h)
                color = (255, 0, 0)
                thickness = 1
                image = cv.rectangle(image, start_point, end_point, color, thickness)

#     plt.figure(figsize=figsize)
    plt.imshow(image)
    plt.show()
```

Fig: code snippet of get_x_y_spacing and plot_bubble_rec_bbox function

Output from phase III:

The output from the phase III is a 3D list of bubble centers with estimated possible values for missing ones. However, we need to analyze how accurately our functions are working, for this particular reason we need to plot those bubble centers with a rectangular bounding box around each bubble. We can analyze the result from below figures

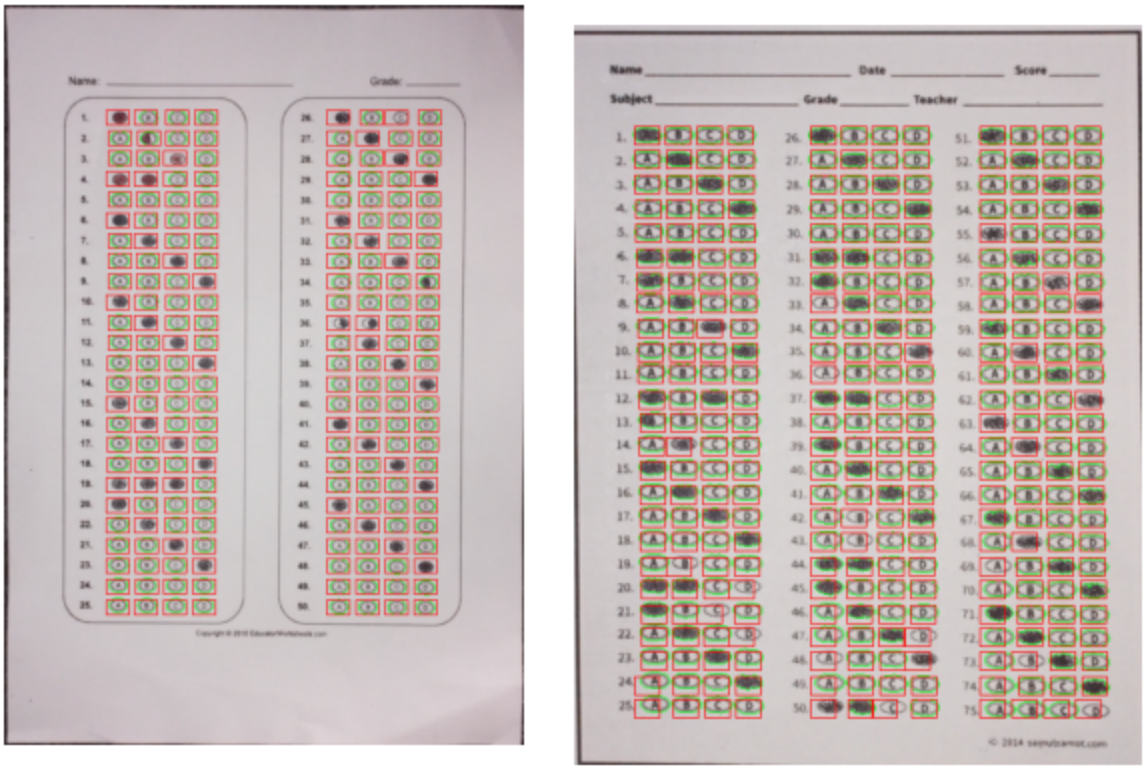


Fig: bubble sheet with detected contours represented by green line and rectangular bounding box around each bubble (detected and estimated one)

Conclusion:

We can clearly see that we have successfully estimated the possible position of the missing bubbles.

PHASE IV: Mark evaluation phase

Sub-phase I: detecting chosen option

In this phase, we will detect the chosen option and use that information for evaluation marks by comparing it with the correct option.

- **Function to detect the chosen option**

```python
def compute_proportion_of_black(image): #image is an binary image
    '''
    Parameter:
        *image : takes an binarized image

    Description:
        It will calculate the percentage of black (0) pixels in that image.

    Returns:
        * proportion of black pixels
    '''
    # convert pixel value 255 into 1
    pixel_ones_count = (np.array(image) % 254).sum()
    total_pixel = image.shape[0] * image.shape[1]

    # calculating the perportion of black in an image
    proportion_of_zeros = (total_pixel - pixel_ones_count)/total_pixel
    return proportion_of_zeros

def crop_area(image):
    '''
    parameter:
        *image: cropped image around the rectangular bounding box around a bubble

    Description:
        It will further cropped the image just to include the bubble, so that we will get more accurate result.

    Returns:
        *cropped image in a numpy array formate.
    '''
    new_rows = []
    # crop row
    for row in np.array(image)%254:
        sum_ = row.sum()
        if sum_ != len(row):
            new_rows.append(list(row))
    # crop columns
    df = pd.DataFrame(new_rows)
    for column in df.columns:
        if sum(df[column]) == len(df[column]):
            df.drop(column, axis=1, inplace=True)
    return np.array(df)

def detect_choice(question_bubble, image, rect_w, rect_h, black_percentage=0.65):
    '''
    Parameters:
        * question_bubble: list of center of bubbles of a particular question
        * image: binarized image of the bubble answer sheet
        * rect_w: width of a rectangular bounding box
        * rect_h: height of a rectangular bounding box
        * black_percentage: minimum percentage of black pixels for a chosen option

    Details:
        It will check each bubbles of a question and evaluate the proportion of black pixels in it,
        based on that value it will determine which option is chosen and also accounts the errors.

    Returns:
        * index of chosen option or -1 if no option is chosen or multiple options are chosen.

    '''
    choice = []
    for index, bubble in enumerate(question_bubble):

        bubble_area = image[round(bubble[1]- 12): round(bubble[1]+ 12),
                            round(bubble[0] - 12): round(bubble[0] + 12)]
        bubble_area = crop_area(bubble_area)
#         plt.imshow(bubble_area, cmap='gray')
#         plt.show()
        proportion_of_zeros = compute_proportion_of_black(bubble_area)
        if proportion_of_zeros >= black_percentage:
            choice.append(index+1)
    if len(choice) == 1:
        return choice[0]
    else:
        return -1
```

Fig: code snippet detect_choice and the helper functions

- **Testing the accuracy**

  To test the accuracy, we will compare the detected chosen option of each question to the corresponding actual chosen option.

```
# checking accuracy of our model
total_questions = len(actual_user_choices)
matched_count = 0
for detected, actual in zip(answers, actual_user_choices):
    if detected == actual:
         matched_count += 1

accuracy = matched_count / total_questions * 100
print(f" accuracy : {accuracy}")
print(f" total questions: {total_questions}")
print(f" matched count: {matched_count}")
```

```
accuracy : 92.0
total questions: 50
matched count: 46
```

Fig: screenshot of the accuracy on bubble answer sheet (sample 1)

```
accuracy : 98.66666666666667
total questions: 75
matched count: 74
```

Fig: screenshot of the accuracy on bubble answer sheet (sample 2)

Future Work:

1. Optimize the code
2. Write a different and more simple approach for the groupby_axis_x function.
3. Transform the code into command line tools that reads the csv file which contains the correct option of each question, evaluate the scanned answer sheets and write the obtained marks into another csv file.