

Computer Assignment 2

مصطفی کرمانی نیا 810101575

امیر نداف فهمیده 810101540

بخش اول) پیش از هر کار دیگری، در ابتدا مپست را لود می‌کنیم:

```
%% Part 0: Loading the mapset

% Get a list of all files in the "Map Set" directory
files = dir('Map Set');

% Calculate the number of valid files (not '.' and '..')
% The first two elements ('.' and '..') are not image files, so delete them.
len = length(files) - 2;

% Initialize a cell array to store training data.
% The first row will store the images, and the second row will store the corresponding labels (characters).
TRAIN = cell(2, len);

% Loop through each file in the directory, starting from the third file (since the first two are '.' and '..')
for i = 1:len
    % Read the image from the file and store it in the first row of the TRAIN cell array
    TRAIN{1, i} = imread([files(i + 2).folder, '\', files(i + 2).name]);

    % Extract the first character of the file name (its the label)
    % and store it in the second row of the TRAIN cell array
    TRAIN{2, i} = files(i + 2).name(1);
end

% Save the TRAIN cell array to a file named "TRAININGSET.mat"
% This will store the images and corresponding labels for future use.
save TRAININGSET TRAIN;
```

- برای اینکار ابتدا از فولدری که عکس‌های اعداد و حروف در آن است، عکس‌ها را با `imread` خوانده (البته دو عضو اول را همانطور که در کامنت کد نوشته ایم، نمی‌خوانیم چون تصویر نبوده و صرفاً علامت `(.)` و `(..)` هستند.) و در یک `cell` بنام `TRAIN` می‌ریزیم. که ردیف اول آن عکس‌هایی با پیکسل‌های باینری و به اندازه 24×42 بوده و ردیف دوم لیبل آن عکس‌ها (همان حرف اول نام فایل هر عکس) است.

Variables - TRAIN								
TRAIN								
2x62 cell								
	1	2	3	4	5	6	7	8
1	42x24 logical	42x24 logic...	42x24 logic...	42x24 logic...	42x24 logic...	42x24 logic...	42x24 logic...	42x24 logic...
2	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'
3								

- سپس این cell را در فایلی انکود شده بنام TRAININGSET.mat می‌ریزیم تا بعداً آنرا لود کرده و استفاده کنیم. (در بخش 7)

1- در این بخش صرفاً عکس پلاک ورودی را از کاربر می‌گیریم:

```
%% part 1 : input picture
[file,path]=uigetfile({'*.jpg;*.png;*.jpeg'},'Select your image');

% build full file path and read the image from the selected file path
img = imread(fullfile(path, file));
figure, imshow(img);
```

- ابتدا با uigetfile یک پنجره جهت انتخاب تصویر به کاربر نمایش می‌دهیم.

- سپس توسط fullfile آدرس دایرکتوری عکس را به اسم خود فایل عکس وصل کرده (میتوانستیم این کار را همانند بخش قبلی بصورت دستی و با یک اسلش گذاشتن و این‌ها انجام دهیم اما این تابع را هم می‌خواستیم امتحان کنیم.) و توسط imread آن تصویر را خوانده و با تولید یک figure و استفاده از imshow آنرا نمایش می‌دهیم که از درستی کارمان مطمئن شویم.

مثلاً:



-2

در این بخش هم با استفاده از تابع `imresize` ابعاد عکس را به ابعاد خواسته شده می‌رسانیم و باز هم جهت اطمینان از درستی عکس حاصل، آنرا نمایش می‌دهیم.

```
%% part 2 : resize  
img = imresize(img, [300, 500]);  
figure, imshow(img);
```

مثلا:



3- در این بخش بدلیل کاهش حجم اطلاعات، تصویر را با رابطه ی داده شده، سیاه و سفید می کنیم.

```
%% part 3 : build grayscale image
function grayImg = mygrayfun(colorImg)
    % Convert the input color image to grayscale using the weighted sum of color channels
    % grayImg = 0.299 * Red + 0.578 * Green + 0.114 * Blue

    % Extract color channels from the input image
    redChannel = colorImg(:,:,1);
    greenChannel = colorImg(:,:,2);
    blueChannel = colorImg(:,:,3);

    % Apply the grayscale formula
    grayImg = 0.299 * redChannel + 0.578 * greenChannel + 0.114 * blueChannel;
end
```

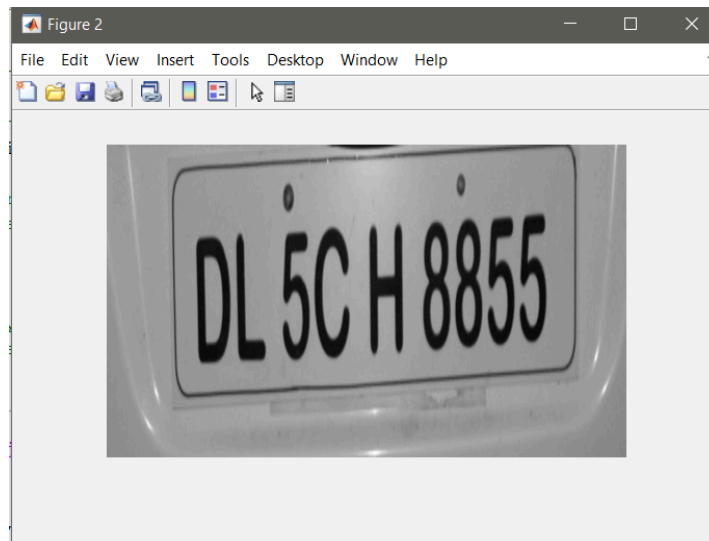
```
%% Convert the color image to grayscale using mygrayfun function
grayImg = mygrayfun(img);
figure, imshow(grayImg);
```

- ساختن تابع mygrayscale که عکس رنگی را گرفته و عکس سیاه و سفید برمیگرداند را بوسیله ی شکستن تصویر اولیه که RGB بود به سه ماتریس دو بعدی نمایانگر مقدار رنگ قرمز، آبی و سبز شروع می کنیم.

- حالا ضرایب خواسته شده توسط سوال را در ماتریس هر رنگ ضرب کرده و سپس هر سه را با هم جمع می کنیم تا طبق فرمولی صورت سوال، به عکس نهایی برسیم.

- جهت استفاده از این تابع هم عکس رنگی را به آن می دهیم و خروجی که عکس سیاه و سفید است در یک ماتریس جدید ریخته و در صورت نیاز، آنرا باز نمایش می دهیم.

مثلا:



4- در این بخش عکس را باینری میکنیم:

```
%% part 4: Convert the grayscale image to binary using a threshold
function binaryImage = mybinaryfun(grayImage, threshold)
    binaryImage = grayImage > threshold; % Convert to binary using the threshold
    binaryImage = ~binaryImage;
end
```

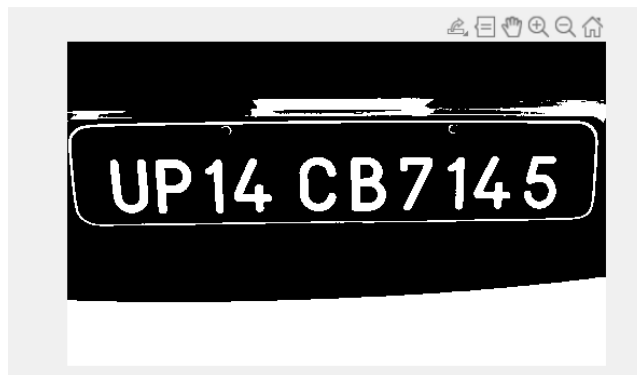
```
%% Convert the grayImg to binary image using mybinaryfun function
threshold = 100; % Selected threshold value
binaryImage = mybinaryfun(grayImg, threshold);
figure, imshow(binaryImage);
```

- برای زدن این تابع از یک ایده و قابلیت های متلب استفاده کرده و برای اعمال threshold صرفا از عبارت شرطی `grayImage > threshold` استفاده کردیم، که برای تک تک پیکسل های آن عکس اعمال شده و یعنی اگر مقدار آن پیکسل از threshold بیشتر بود، این عبارت شرطی عدد 1 را بمعنی true برمیگرداند و ما هم آن را به آن پیکسل منسوب میکنیم که بمعنای رنگ سفید است و بلعکس، اگر آن پیکسل مقدارش از عدد threshold کمتر بود که یعنی از یک حدی سیاهتر بود، این عبارت شرطی مقدار 0 یا false برمیگرداند که بمعنی رنگ سیاه در یک عکس باینری است.

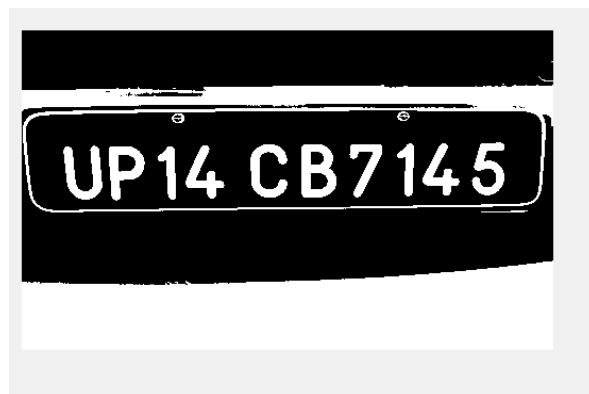
- در نهایت تمام مقادیر را معکوس میکنیم، چون در حالت عادی اعداد روی پلاک، به رنگ مشکی هستند که باعث گرفتن مقدار 1 در این تبدیل میشود، اما اعداد موجود در دیتامپ ما، سفید هستند، پس اگر بیت های عکس را همینجا معکوس کنیم تا اعداد روی پلاک به رنگ سفید و با مقدار 1 نمایش داده شوند، بعدا در correlation کارمان راحتتر و سر راست میشود.

- چون تصویر نوشته های پلاک معمولا خیلی سیاه است پس اگر threshold را کمتر از 128 بگیریم هم مشکلی رخ نمیدهد پس اینکار را کردیم تا بخشی از نویز ها در همین مرحله رسماً حذف شوند:

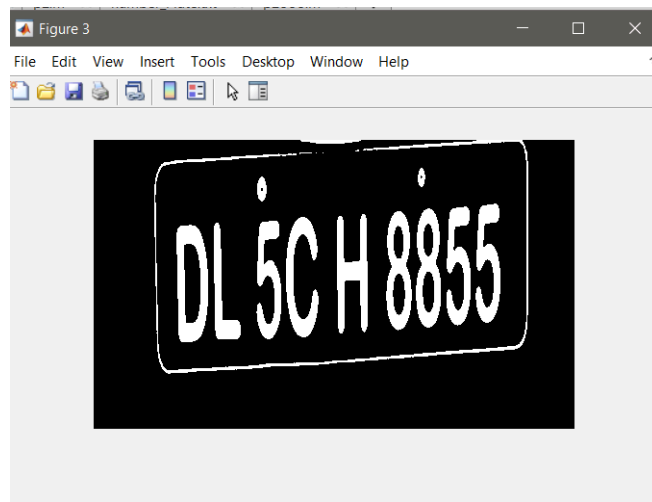
با 100 threshold:



- با 128 threshold:



مثلا:



5- در این بخش نویزهای تصویر (کامپوننت های بسیار کوچک) را حذف میکنیم. چون تابع طولانی است، آنرا تقسیم کرده ام و توضیح میدهم.

```
%% part 5: Remove small connected components (noise) without using bwareaopen
function cleanImage = myremovecom(binaryImage, n_minSize)
    % Step 1: Find all points where the picture is 1 (white pixels in the binary image)
    [row, col] = find(binaryImage == 1); % Get row and column indices of white pixels (value 1)
    POINTS = [row'; col']; % Store points as a 2*N matrix (row in first row, col in second row)

    if isempty(POINTS)
        cleanImage = binaryImage; % If no points are found, return the original image
        return;
    end

    cleanImage = zeros(size(binaryImage)); % Initialize the clean image (binary)

    % Step 2: Process all points and group them into connected components
    while ~isempty(POINTS)
        % Initialize the current object with the first point
        initpoint = POINTS(:, 1);
        POINTS(:, 1) = []; % Remove the used point

        % Use a stack to explore the connected component
        points_to_explore = initpoint;
        currentObject = initpoint;
```

- این تابع در ابتدا لوکیشن بخش هایی از تصویر را که مقدار 1 دارند (سفیدند) پیدا میکند و دو وکتور به نام row و col می‌ریزد. سپس ما این دو وکتور را به هم چسبانده و یک ماتریس با دو ردیف تشکیل میدهم که هر ستون آن نمایانگر شماره‌ی سطر و ستون یکی از نقاط سفید رنگ است.

- سپس برای اطمینان فقط بررسی میکنیم که آیا اصلا عکس ما نقطه‌ی سفیدی داشته است یا نه، اگر نداشته بود کلا همان تصویر را برمیگردانیم. مگر نه، در وهله‌ی اول، عکس نهایی را بصورت یک ماتریس از 0 ها و به سبب عکس اولیه تعریف میکنیم. (یعنی یک تصویر کاملا سیاه داریم)

- حالا در یک حلقه‌ی while قرار است به روش هایی که گفته میشوند، دانه به دانه کامپوننت های عکس را یافته و اگر آن کامپوننت از یک حدی بزرگتر بود، آنرا مورد تایید قرار داده و به عکس نهایی اضافه کنیم، و هر نقطه‌ای که به یک کامپوننت اضافه میکنیم، آنرا از لیست نقاط اولیه حذف میکنیم و این حلقه‌ی بیرونی، تا وقتی تمام نقاط به کامپوننت های مربوط به خودشان اضافه شوند، ادامه خواهد داشت.

- حلقه ی بیرونی به این شکل آغاز میشود که نقطه ی اول از لیست نقاط سفید را برداشته و آنرا از لیست نقاط سفید حذف کرده و به لیست نقاط کامپوننت اول اضافه میکنیم. یک استک هم می سازیم بنام `points_to_explore` که قرار است بوسیله ی آن و الگوریتم DFS تمام نقاط سفید مربوط به این کامپوننت را بازدید کنیم.

```
] while ~isempty(points_to_explore)
    % Take the first point from the stack
    current_point = points_to_explore(:, 1);
    points_to_explore(:, 1) = []; % Remove the used point

    % Find new points connected to the current_point
    [POINTS, newPoints] = close_points(current_point, POINTS);

    % Add the new points to the current object and exploration stack
    currentObject = [currentObject newPoints];
    points_to_explore = [points_to_explore newPoints];
- end

% If the connected component's size is large enough, add it to the clean image
if size(currentObject, 2) >= n_minSize
    for j = 1:size(currentObject, 2)
        cleanImage(currentObject(1, j), currentObject(2, j)) = 1;
- end
end
- end
end
```

- در حلقه ی دوم، تا وقتی استک ما خالی نشده است هر بار عضو اول استک نقاط بازدید نشده را برداشته و همسایه های سفید و مجاور آن را بوسیله ی تابع `close_points` (که در انتها شرح میدهم آنرا) پیدا کرده و آنها را به کامپوننت فعلی اضافه کرده و در استک بازدید نشده ها قرار می دهیم تا در iteration های بعدی، همسایه های آنها را هم به لیست اضافه کنیم.

- پس نهایتا استک نقاط بازدید نشده خالی میشود که یعنی دیگر نقطه ی سفیدی نیست که بتوان از طریق همسایگی، به کامپوننت فعلی افزود، پس از حلقه ی داخلی بیرون آمده و بررسی میکنیم که اگر تعداد نقاط این کامپوننت که تکمیل شده است، از مقدار مینیمم تعیین شده بیشتر بود، آنرا به عکس نهایی اضافه کنیم و اگر کمتر بود آنرا دور میریزیم.

- نهایتاً دوباره به حلقه ی بیرونی رفته و بین نقاط سفید باقی مانده دوباره کامپوننت پیدا میکنیم و همینطور میرویم تا تمام نقاط وارد کامپوننت هایی شده و یا وارد عکس نهایی شوند و یا دور ریخته شوند.

```
%% Helper function to find points close to a given point
function [remainingPoints, neighbor_Points] = close_points(initpoint, POINTS)
    % This function returns points in POINTS that are close to
    % the initpoint and remove them from POINTS

    neighbor_Points = [];
    remainingPoints = POINTS;
    for i = size(POINTS, 2):-1:1
        % Check the distance between the initpoint and the current point
        if abs(POINTS(1, i) - initpoint(1)) <= 1 && abs(POINTS(2, i) - initpoint(2)) <= 1
            % Add the point to newPoints and remove it from remainingPoints
            neighbor_Points = [neighbor_Points POINTS(:, i)];
            remainingPoints(:, i) = [];
        end
    end
end
```

- عملکرد تابع close_opoints هم به این گونه است که نقطه ی مدنظر و لیست تمام نقاط سفید را به آن میدهم و آن از بین 8 همسایه ی نزدیک به آن نقطه جستجو کرده و می بیند کدام نقاط سفیدند. سپس همسایه های سفید را داخل لیست neighbor_points ریخته و آنها را از لیست نقاط سفید اولیه حذف میکند.

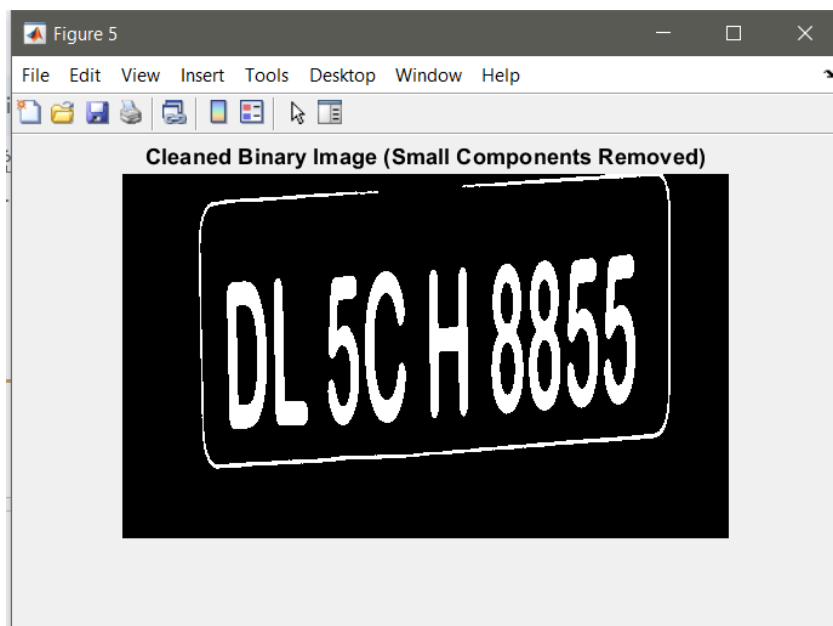
- دلیل استفاده از حلقه ی معکوس هم این است که چون در طی این حلقه اگر همسایه ای یافت شود، از لیست remaining_points که در ابتدا مساوی با همان points بوده است، حذف میشود و حلقه هم در حال لوپ زدن تا رسیدن به آخرین ایندکس points است پس اگر حلقه ی مستقیم بزنیم ممکن است به مشکلات ایندکسی بخوریم.

- نهایتاً بعنوان خروجی لیستی از همسایگان سفید نقطه ی ورودی، به همراه نقاط سفید باقی مانده برمیگرداند.

```
%% the cleaned binary image
minSize = 300; % Minimum size of objects to keep
cleanImage = myremovecom(binaryImage, minSize);
figure, imshow(cleanImage);
title('Cleaned Binary Image (Small Components Removed)');
```

- برای استفاده از این تابع هم کفایت تصویر باینری شده و مینیمم سائز معتبر برای یک کامپوننت را به آن بدهیم و تصویر بدون نویز را دریافت کنیم.

مثلا:



6- در این بخش تصویر را segment بندی میکنیم.

```
%% part 6: segmenting the picture
function [labeledImage, numObjects] = mysegmentation(cleanImage)
    % Step 1: Find all points where the picture is 1 (white pixels in the clean binary image)
    [row, col] = find(cleanImage == 1); % Get row and column indices of white pixels (value 1)
    POINTS = [row'; col']; % Store points as a 2*N matrix (row in first row, col in second row)

    if isempty(POINTS)
        labeledImage = zeros(size(cleanImage)); % If no points are found, return an empty labeled image
        numObjects = 0; % No objects detected
        return;
    end

    labeledImage = zeros(size(cleanImage)); % Initialize the labeled image
    currentLabel = 0; % Start with label 0

    % Step 2: Process all points and group them into connected components
    while ~isempty(POINTS)
        % Increment the label for the new object
        currentLabel = currentLabel + 1;

        % Initialize the current object with the first point
        initpoint = POINTS(:, 1);
        POINTS(:, 1) = []; % Remove the used point

        % Use a stack to explore the connected component
        points_to_explore = initpoint;
        currentObject = initpoint;
```

- کد این بخش از خیلی لحاظ شبیه بخش قبل است. چون رسماً بار باید کامپوننت ها را با DFS یا الگوریتم های دیگر تشخیص اما اینبار به هر کدام لیبل جداگانه میزنیم.

- پس مجدداً با یافتن نقاط سفید شروع کرده و تصویر اولیه را یک تصویر کاملاً سیاه در نظر گرفته و حلقه ی بیرونی که تا تمام شدن تمام نقاط سفید ادامه دارد را ساخته و حلقه ی درونی که تا خالی شدن لیست نقاط بازدید نشده در استک ادامه دارد را ساخته و در حلقه ی درونی هر بار همسایه های نقطه ی مدنظر (که بوسیله ی تابع `close_points` که قبلاً توضیح دادیم، پیدا میکنیم) را به استک نقاط بازدید نشده اضافه کرده و هر بار که همسایه های یک نقطه را میبینیم خود آن نقطه را از استک حذف میکنیم و نهایتاً طبق الگوریتم مشابه DFS تمام نقاطی که میتوانند توسط همسایگی به یک کامپوننت متصل شوند، به آن متصل میشوند.

```

while ~isempty(points_to_explore)
    % Take the first point from the stack
    current_point = points_to_explore(:, 1);
    points_to_explore(:, 1) = []; % Remove the used point

    % Find new points connected to the current_point
    [POINTS, newPoints] = close_points(current_point, POINTS);

    % Add the new points to the current object and exploration stack
    currentObject = [currentObject newPoints];
    points_to_explore = [points_to_explore newPoints];
end

% Step 3: Label the connected component in labeledImage
for j = 1:size(currentObject, 2)
    labeledImage(currentObject(1, j), currentObject(2, j)) = currentLabel;
end
end

% Return the labeled image and the number of objects
numObjects = currentLabel; % Total number of labeled objects
end

```

- تفاوت کوچک این بخش با بخش قبلی وقتی شروع می شود که ما یک کامپوننت را بطور کامل یافتیم. پس از حلقه ی درونی خارج شده و اینجا لیبل ی که در حلقه ی بیرونی با اتمام هر کامپوننت یک واحد به آن اضافه می کنیم را به لوکیشن مربوط به تک تک نقاط آن کامپوننت نسبت می دهیم.

- نهایتاً هم یکی از خروجی های تابع، عکس لیبل زده شده است، که در آن تمام اعضای هر کامپوننت لیبل یکسان داشته ولی هر کامپوننت با دیگری لیبلش فرق میکند. و تعداد نهایی کامپوننت ها یا عبارتی لیبل ها هم خروجی داده میشود.

```

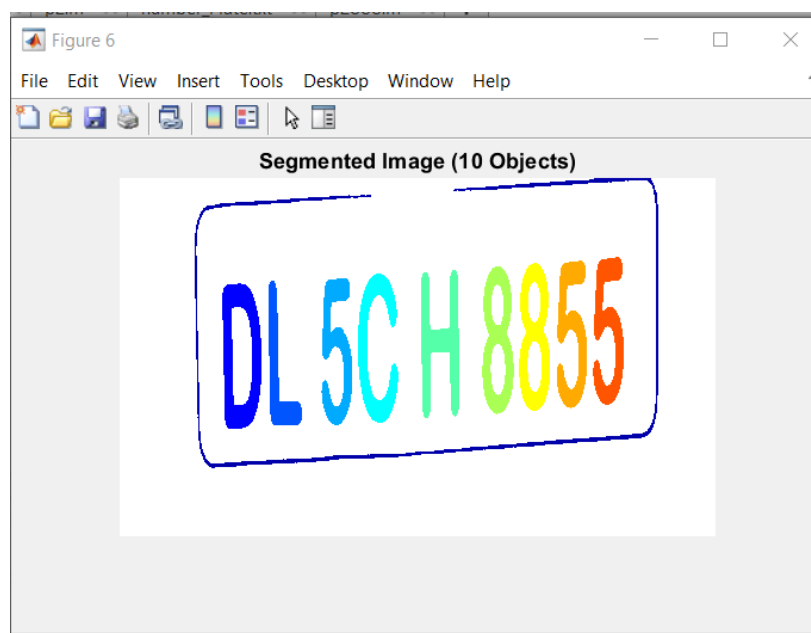
%% the segmented (labeled) image
[labeledImage, numObjects] = mysegmentation(cleanImage);
figure, imshow(label2rgb(labeledImage)); % Convert labels to colors for visualization
title(['Segmented Image (' , num2str(numObjects), ' Objects)']);

```

- نکته اینجاست که تصویر نهایی، یک تصویر باینری نیست و فقط یکسری پیکسل ها دارد که لیبل خورده اند. پس بطور مستقیم قابل رسم نیست، پس برای دیده شدن نتیجه ی کار این تابع، از تابعی کمکی بنام

label2rgb استفاده کردیم که هر لیبل را به یک رنگ rgb تبدیل کرده و بدین ترتیب میتوان عملیات لیبل زدن را قابل مشاهده کرد.

مثلا:



7- تصمیم گیری نهایی

```
%% Part 7: Compact Decision-Making Using Correlation (corr2)

% Load the saved training set
load TRAININGSET;
numTemplates = size(TRAIN, 2);

recognizedText = ''; % Store recognized characters
for segmentIndex = 1:numObjects
    [rowIndices, colIndices] = find(labeledImage == segmentIndex); % Find row and column indices of the current object

    % Extract the sub-image (the current segment) from the binary cleanImage.
    % The segment is defined by the bounding box surrounding the object (min/max row and column indices).
    % Then, resize the extracted segment to 42x24 pixels to match the size of the templates in the training set.
    currentSegment = imresize(cleanImage(min(rowIndices):max(rowIndices), min(colIndices):max(colIndices)), [42, 24]);

    % Compute correlation scores for all templates
    ro = zeros(1, numTemplates); % Initialize correlation scores array
    for templateIndex = 1:numTemplates
        ro(templateIndex) = corr2(TRAIN{1, templateIndex}, currentSegment); % Correlation for each template
    end
```

- در ابتدا دیتاستی که در مرحله ی اول سیو کرده بودیم را لود میکنیم. سپس یک رشته ی خالی برای ثبت نتیجه ی نهایی پلاک خوانی تولید میکنیم.

- حالا در یک حلقه for تک تک segment های یافت شده را از روی لیبل آنها در تصویر لیبل دار پیدا کرده (به این صورت که ستون ها و ردیف های آنها را در قالب دو وکتور دریافت میکنیم با تابع find سپس از کمترین سطری که آن لیبل را دارد تا بزرگترین سطری که آن لیبل را دارد و از کوچکترین تا بزرگترین ستونی که آن لیبل را دارند را از تصویر بدون نویز انتخاب کرده و رسماً انگار یک مربع دور هر segment میکشیم) و سپس resize میکنیم تا هم اندازه با عکس های موجود در دیتاست شوند.

- حالا یک وکتور به اندازه ی تعداد اعضای دیتاست در نظر میگیریم تا با یک حلقه ی for درونی مقدار کورلیشن تک تک اعضای دیتاست را با آن segment ای که مدنظر است، بیابیم. و عملیات کورلیشن را با تابع corr2 میکنیم.

```
% Create a new figure for each segment with 2 subplots: one for the image, one for the plot
fig = figure; % Create a new figure for every segment

% Subplot 1: Show the image of the segment
subplot(1, 2, 1); % 1 row, 2 columns, 1st subplot
imshow(currentSegment); % Display the segment image
title(['Segment ', num2str(segmentIndex)]);

% Subplot 2: Show the correlation scores (ro)
subplot(1, 2, 2); % 1 row, 2 columns, 2nd subplot
bar(ro); % Bar plot of correlation scores
title(['Correlation Scores for Segment ', num2str(segmentIndex)]);
xlabel('Template Label');
ylabel('Correlation Score');

% Customize the x-axis to display the template labels instead of indices
xticks(1:numTemplates); % Set x-ticks at positions corresponding to each template
xticklabels(TRAIN(2, :)); % Set x-tick labels to the corresponding template labels (characters)

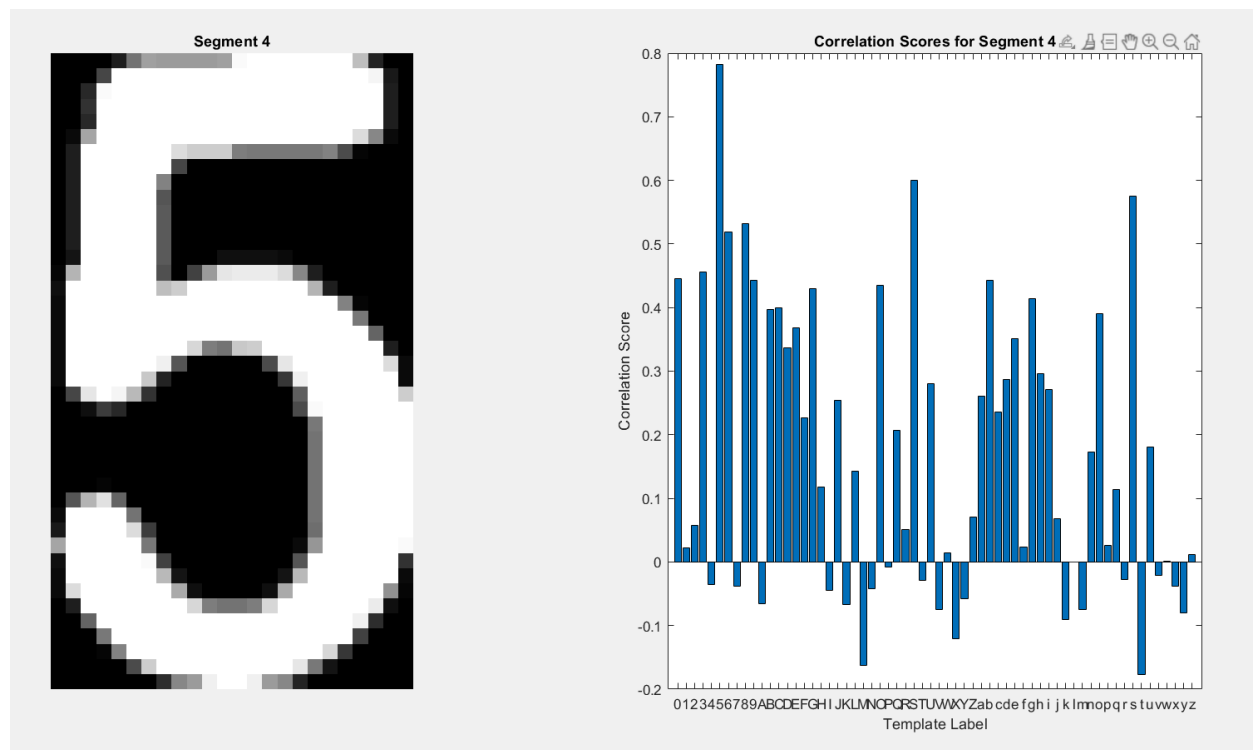
drawnow; % Ensure that the figure is rendered before the next iteration

% Find the best match and apply a threshold
[bestScore, bestMatchIndex] = max(ro); % Get the best match score and index
if bestScore > 0.48 % Threshold for accepting a match
    recognizedText = [recognizedText, TRAIN{2, bestMatchIndex}]; % Append detected character
end
end
```

- حالا برای هر segment یک figure میسازیم تا ضمن نشان دادن تصویر خود آن کامپوننت، نموداری شامل مقدار کورلیشن آن کامپوننت با تک تک اعضای مپ ست به ما نمایش بدهد و البته محور افقی را طوری تنظیم میکنیم که خود حرف یا عددی که عکس آن در مپ ست است را برایمان بنویسد تا نمودار قابل فهم تر باشد.

- سپس عکسی که بالاترین مقدار کورلیشن را داشت انتخاب میکنیم و بعنوان حدس خود از حرف یا عدد آن کامپوننت اعلام میکنیم و به رشته ی مربوط به حدس نهایی اضافه میکنیم و به سراغ کامپوننت بعدی می رویم. در این میان چون ممکن است عکسمان در مرحله ی denoise بطور کامل هم denoise نشده باشد. یک threshold میگذاریم که اگر مقدار کورلیشن هیچ کدام از اعضای مپ ست با یک کامپوننت خاص، از مقدار آن بیشتر نشد، کلا آن را بعنوان نویز یا کامپوننت ناخوانا دور بریزد.

مثلا figure مربوط به یکی از بخش های پلاکی که داشتیم:



- همانطور که مشخص است، مقدار کورلیشن برای عدد 5 از بقیه اعداد و حروف مپ ست بیشتر بوده و در نتیجه این segment به درستی تشخیص داده میشود

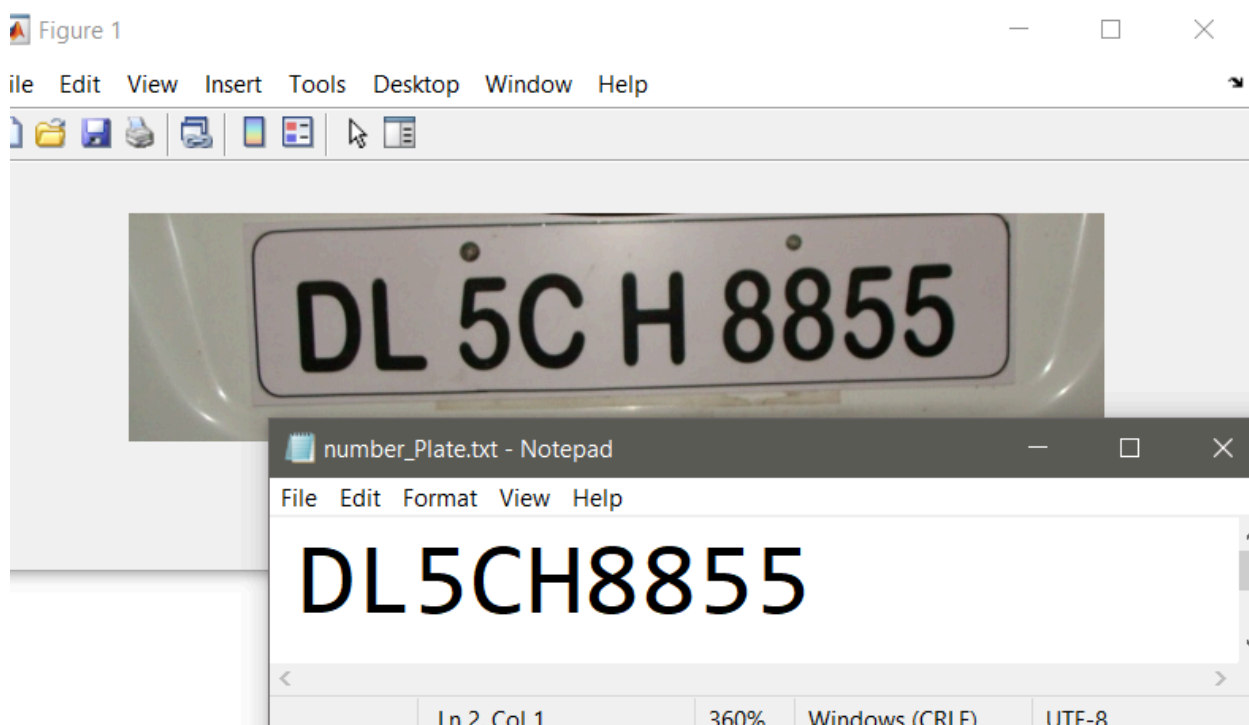
- دلیل منفی بودن مقدار کورلیشن در برخی مقادیر هم این است که تابع corr2 مقدار میانگین را هم از حاصل کرلیشن کم کرده تا آنرا تا حدی متعادل کند.

8- ذخیره ی نتیجه

```
%% Part 8: Save the Detected Characters to a File  
% Display the result  
disp(['Detected pelaak: ', recognizedText]);  
  
% Save the detected characters to a file  
fileID = fopen('number_Plate.txt', 'wt');  
fprintf(fileID, '%s\n', recognizedText);  
fclose(fileID);  
winopen('number_Plate.txt');
```

- نهایتاً نتیجه را چاپ کرده و در یک فایل txt نوشته و فایل را می بندیم و یکبار هم جهت بررسی نهایی، آن فایل را open میکنیم.

نتیجه ی نهایی در این مثال:



- همانطور که مشخص است، عدد و حروف ها به درستی تشخیص داده شده اند.

بخش دوم)

1. چالش اول این بخش، ساختن دیتاست فارسی است. ما با جستجوی بسیار در اینترنت (و حتی پرداخت 4 هزار تومن پول!) نهایتاً متوجه شدیم که شبیه ترین فونت فارسی به فونت به کار رفته در پلاک ها، فونت B Titr است. پس به کمک مقداری کدهای پایتون که در نوت بوک persian_DS_generator سیو شده است، یک دیتاست بر اساس فونت B Titr ساختیم.

```
[ ] !pip install pillow
```

Requirement already satisfied: pillow in /usr/local/lib/python3.10/dist-packages (10.4.0)

```
[ ] from google.colab import files
    uploaded = files.upload() # آپلود کردن فایل فونت، با نام B_Titr.ttf
```

Show hidden output

```
from PIL import Image, ImageDraw, ImageFont
import shutil
```

```
# مسیر فونت آپلود شده
font_path = "/content/B_Titr.ttf"
```

```
[ ] # سایز و رنگ فونت
    font_size = 100
    font_color = (0, 0, 0) # مشکی
    background_color = (255, 255, 255) # سفید

    # ایجاد فونت
    font = ImageFont.truetype(font_path, font_size)

    # لیست اعداد فارسی و حروف مورد نظر
    characters = [str(num) for num in range(10)] + ['ب', 'ج', 'س', 'ص', 'ط', 'ز', 'ل', 'م', 'ن', 'و', 'ه', 'ی']
```

```
# ساخت تصویر برای هر کاراکتر (عدد یا حرف)
for char in characters:
    # اندازه تصویر
    img_size = (150, 150)

    # ایجاد تصویر جدید
    img = Image.new("RGB", img_size, background_color)
    draw = ImageDraw.Draw(img)

    # اندازه متن
    text_bbox = draw.textbbox((0, 0), char, font=font)
    text_width = text_bbox[2] - text_bbox[0]
    text_height = text_bbox[3] - text_bbox[1]

    # مرکز کردن متن در تصویر
    text_position = ((img_size[0] - text_width) // 2, (img_size[1] - text_height) // 2)

    # کشیدن کاراکتر روی تصویر
    draw.text(text_position, char, font=font, fill=font_color)

    # ذخیره تصویر به نام کاراکتر (عدد یا حرف)
    img.save(f"{char}.png")
```

```
[ ] # zip شامل همه تصاویر با استفاده از دستور ZIP ایجاد فایل
    !zip -r characters_images.zip /content/*.png

    # ZIP دانلود فایل
    files.download("characters_images.zip")
```

Show hidden output

- نحوه ی کار کد هم اینجوری است که ابتدا فونت مدنظر را آپلود کرده و سایز و رنگ فونت و پس زمینه را انتخاب میکنیم، که من مثلا متنی مشکی روی زمینه ی سفید میخوام. سپس بکمک کتابخانه ی pillow یک تصویر به سایز دلخواه مثلا 150*150 ایجاد کرده و کاراکتر های مورد نظر را با فونتی که آپلود

کرده ایم فایل tff. آنرا، می نویسیم به ترتیب. حالا توسط تابع boundary box موقعیت مکانی گوشه های چپ و راست و بالا و پایین آن حرف یا عدد موجود در عکس را یافته و سپس طول و عرض آن را بوسیله ی همین مقادیر یافته و آنرا به مرکز تصویر می آوریم. حالا تمام تصاویر را زیپ کرده و دانلود میکنیم.

- حالا وقت لود کردن دیتاست به داخل متلب است:

```
%% Part 0: Loading the mapset

% Get a list of all files in the "Map Set" directory
files = dir('persian_dataset');

% Calculate the number of valid files (not '.' and '..')
% The first two elements ('.' and '..') are not image files, so delete them.
len = length(files) - 2;

% Initialize a cell array to store training data.
% The first row will store the images, and the second row will store the corresponding labels (characters).
TRAIN = cell(2, len);

% Loop through each file in the directory, starting from the third file (since the first two are '.' and '..')
for i = 1:len
    % Read the image from the file and store it in the first row of the TRAIN cell array
    TRAIN{1, i} = imread([files(i + 2).folder, '\', files(i + 2).name]);

    % Extract the first character of the file name (its the label)
    % and store it in the second row of the TRAIN cell array
    TRAIN{2, i} = files(i + 2).name(1);
end
```

- چون این بخش از کد دقیقا مثل سوال اول است، توضیحی نمیدهیم چون صرفا تک تک عکس های مپ ست، لود شده اند در یک cell بنام train. اما مشکل اینجاست که دیتاست ساخته شده توسط پایتون دو مشکل دارد، اولی اینکه تصاویر آن سیاه و سفیدند اما در فرمت RGB هستند و هم 3 بعدی هستند و هم مقادیر پیکسل هایشان باینری نیست، از طرف دیگر ما تصویر حروف و اعداد را در پایتون صرفا به مرکز آوردیم، اما برای کورلیشن بهتر، لازم است دقیقا هر حرف و عدد به لبه های تصویر بچسبد و در حد توان، بزرگ باشد.

Variables - TRAIN

TRAIN

2x25 cell

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	150x150x3 ...	150x150x3 ...	150x150x3 ...	150x150x3 ...	150x150x3 ...	150x150x3 ...	150x150x3 ...	150x150x3 ...	150x150x3 ...	150x150x3 ...	113x150x3 ...	150x150x3 ...	150x150x3 ...	150x150x3 ...	150x150x3 ...
2	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'	'ب'	'ب'	'ج'	'س'	'ص'
3															

```
%% part 0.1 : convert RGB mapset to Binary images and save them again
% Define the threshold value for binarization
threshold = 128; % Adjust the threshold as needed

% Loop through each image in the TRAIN dataset
for i = 1:size(TRAIN, 2)
    % Extract the RGB image from the training set
    rgbImage = TRAIN{1, i};

    % Convert the RGB image to grayscale using your custom function
    grayImage = mygrayfun(rgbImage);

    % Convert the grayscale image to binary (logical) using your custom function
    binaryImage = mybinaryfun(grayImage, threshold);

    % Replace the RGB image in TRAIN with the binary image
    TRAIN{1, i} = binaryImage;
end
```

- پس اولاً به کمک توابعی که خودمان ساختیم در بخش اول، تصویر های مپ ست را ابتدا سیاه و سفید و سپس با یک threshold خوب، باینری کردیم.

```

%% part 0.2 : Code to Detect the Bounding Box, Crop, and Resize

% Loop through each image in the TRAIN dataset
for i = 1:size(TRAIN, 2)
    % Extract the binary image from the training set
    binaryImage = TRAIN{1, i};

    % Find the rows and columns that contain the value 1
    [rows, cols] = find(binaryImage == 1);

    % If no '1' pixels are found, keep the image as is
    if isempty(rows) || isempty(cols)
        croppedImage = binaryImage; % No cropping needed, no '1' found
    else
        % Detect the smallest bounding box that contains all the '1' pixels
        minRow = min(rows);
        maxRow = max(rows);
        minCol = min(cols);
        maxCol = max(cols);

        % Crop the image to that bounding box
        croppedImage = binaryImage(minRow:maxRow, minCol:maxCol);
    end

    % Resize the cropped image back to 150x150
    resizedImage = imresize(croppedImage, [150 150]);

    % Overwrite the original image in the TRAIN cell array with the resized image
    TRAIN{1, i} = resizedImage;
end

% Save the modified TRAIN cell array back to the same TRAININGSET.mat file
save TRAININGSET TRAIN;

```

- ثانیا قرار است آنها را تا حد ممکن برش بدهیم تا حروف و اعداد دقیقا تا لبه های تصویر بیایند. پس بکمک find ابتدا جاهایی که مقدار 1 دارند (سفید هستند) را شناسایی کرده و اگر کلا هیچ بخش سفیدی نبود که همان تصویر را برمیگردانیم چون تصویر کاملا سیاه است. اما اگر بخش های سفید داشت، تصویر را بکمک یافتن کوچکترین و بزرگترین ستون و سطر هایی که سفیدند، برش میزنیم و نهایتا هم resize کرده تا به سایز عکس های مپ ست برسند. نهایتا هم دیتا ست بدست آمده را سیو میکنیم در فایل mat.

- برای مراحل بعدی کار، رسما از همان کدهای بخش اول استفاده کردیم، اما تفاوتی بود که ذکر میکنیم:

- اولین تفاوت این بود که اکنون، دیتاستی که داشتیم 150*150 بود بجای 24*42.
- دومین تفاوت این بود که برخی حروف فارسی، نقطه دارند، و چون نقطه ها جدا از بدنه ی حرف هستند، نقطه و حرف بعنوان کامپوننت های جداگانه شناخته میشد و چون نقطه ها شبیه عدد 0 فارسی هستند، پس تشخیص های اشتباه رخ میداد. برای حل این مشکل با توجه به اینکه این حروف

تعداد کمی داشتند و فارغ از نقطه ای که داشتند، فرم خاصی هم داشتند، متوجه شدیم میتوان بدون در نظر گرفتن نقطه هم آنها را تشخیص داد.

- پس در وهله ی اول، در دیتاست اولیه، علاوه بر حروف نقطه دار، معادل بدون نقطه ی آنها را هم با همان لیبل قرار دادیم

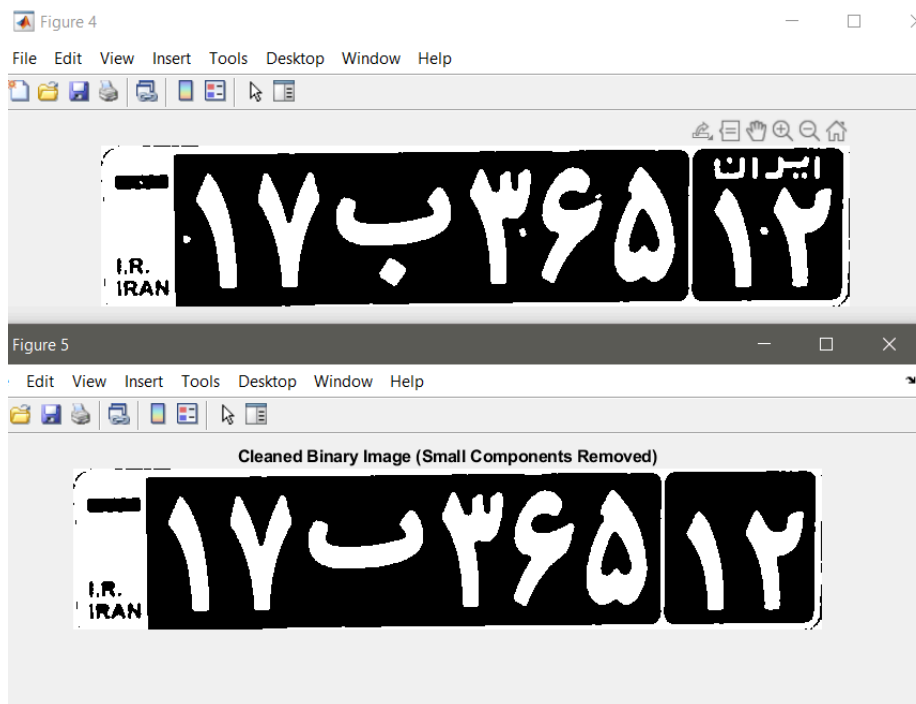
مثلا:



این کار مشکلی پیش نمی آورد چون در بین 12 حرف گفته شده برای تشخیص، هیچ دو حرفی نبودند که فقط در نقطه اختلاف داشته باشند.

- خب تا اینجا مشکل تشخیص حروف نقطه دار، حل شد، اما با مشکل تشخیص نقطه هایشان بعنوان عدد 0 چه کنیم؟ برای اینکار صرفا لازم بود مقدار threshold مدنظر برای denoise را آنقدر بالا ببریم که هم تک نقطه هایی مثل نقطه ی (ب) و هم دونقطه های چسبان مثل نقطه های (ق) بعنوان نویز تشخیص داده شده و حذف بشوند.

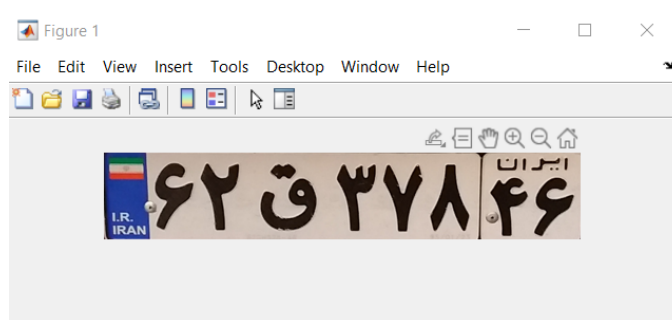
مثلا:



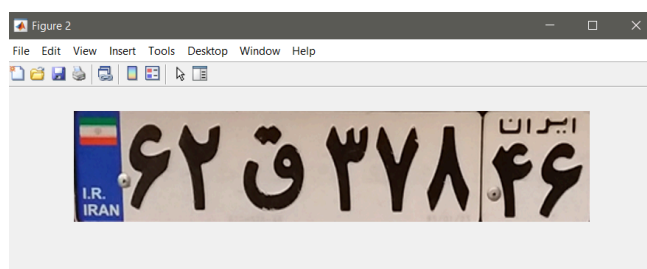
- نکته ی قابل توجه بعدی، فرمی بود که حرف (ه) نوشته میشد در پلاک ها که برخلاف فرم معمولی، به صورت چسبان نوشته می شد که این را هم در کد پایتون اولیه اعمال کردیم و مشکل حل شد.

پس نهایتاً مثلاً برای یک پلاک خاص، داریم:

عکس اصلی:



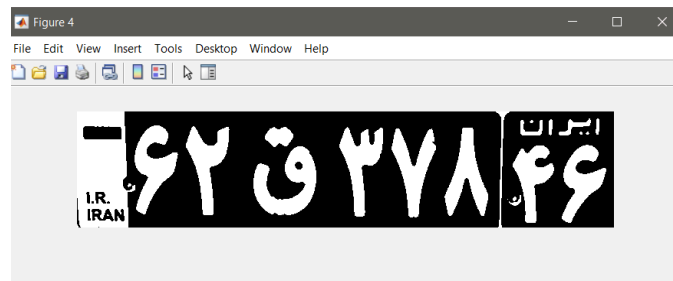
بعد از resize برای شبیه شدن به طول و عرض عدد ها و حروف در عکس های دیتاست:



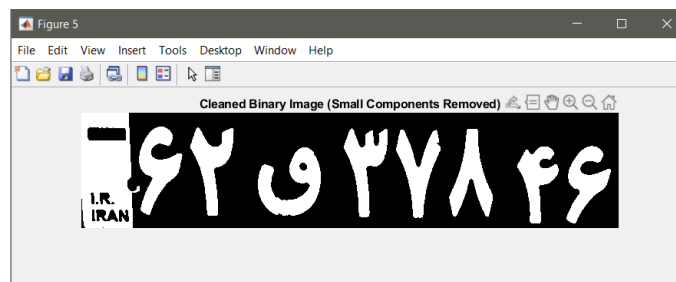
سیاه و سفید کردن:



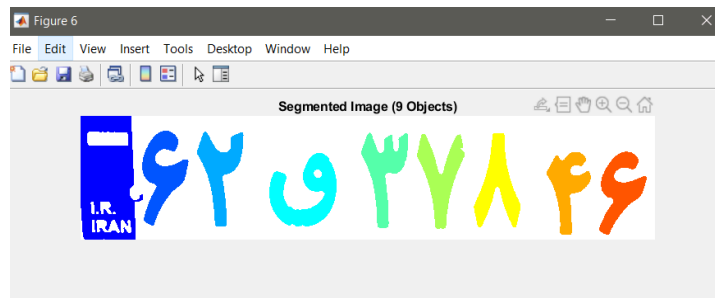
باینری کردن:



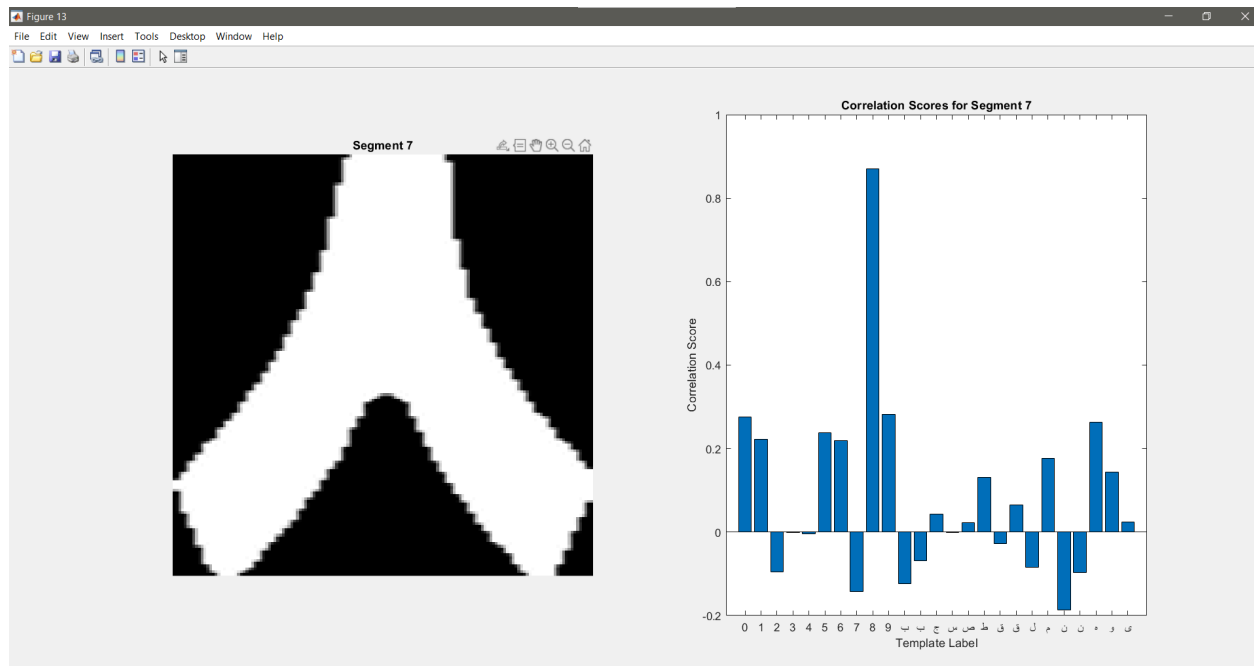
از بین بردن نویز ها و نقطه ی حروف:



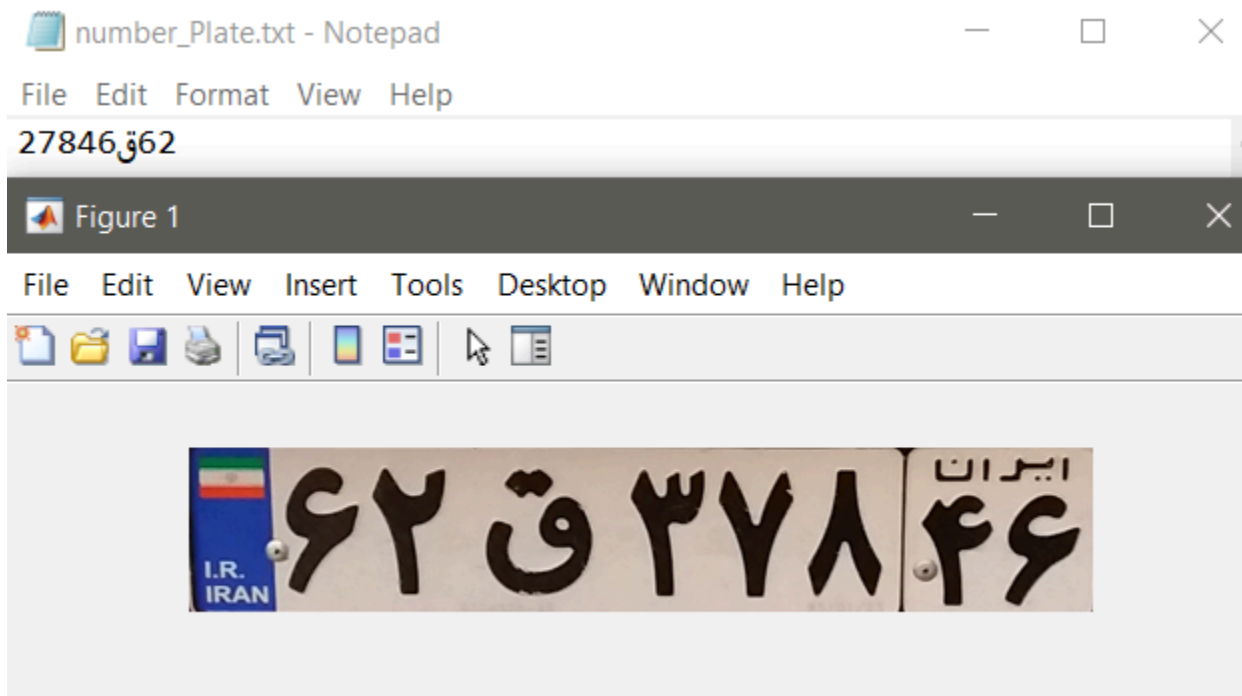
لیبل زدن به segment ها:



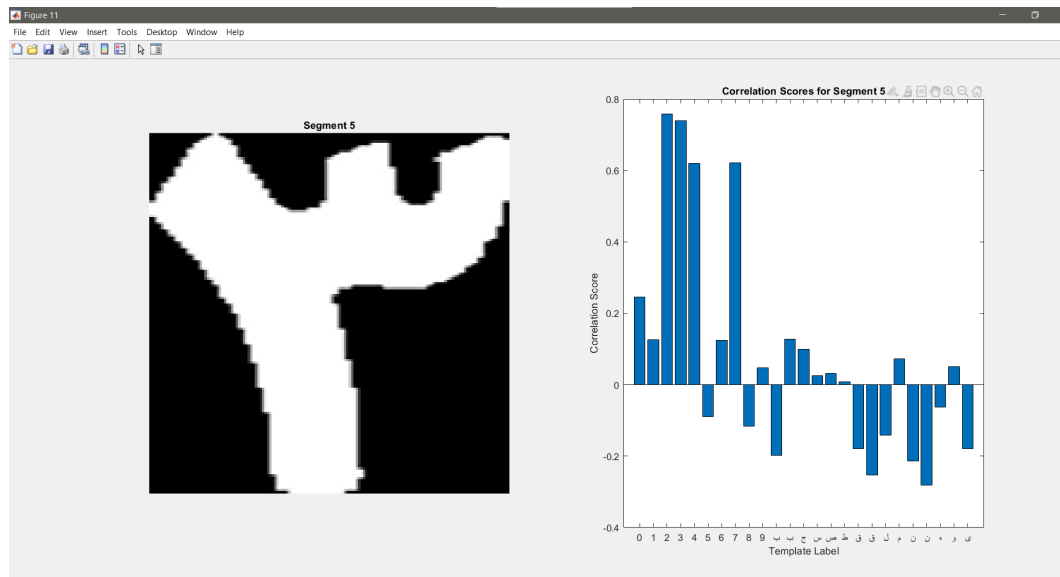
مثالی از تشخیص درست یکی از اعداد:



نتیجه ی نهایی:



نکته ی مهم اینکه چون حروف فارسی در note pad جابه جا میشوند اینگونه نمایش داده می شود، مگر نه از بین 62 و قاف و 378 و 46 فقط عدد 3 در 378 اشتباها 2 تشخیص داده شده که آن هم با فاصله ی کمی در کورلیشن بوده که میتوان بکمک دقیق تر کردن بررسی ها و فیلتر ها، مشکلش را حل کرد:



بخش سوم)

برای این بخش ما از الگوریتم های زیر استفاده کردیم:

- گرفتن تفاضل بین عکس، و عکسی که یک پیکسل به چپ یا راست رفته برای یافتن لبه های عمودی. و همین کار برای یافتن لبه های افقی
- یافتن بخش آبی کنار پلاک و یافتن کل پلاک توسط تناسب طول و عرض پلاک
- گرفتن تفاضل بین عکس و نمونه های شیفته شده با راست یا چپ به مقدار کم، برای فهمیدن منطقه ای با بیشترین تغییرات بین رنگ های خالص سیاه و سفید یا بعبارتی مناطقی با ماکسیمم مشتق رنگی

- از بین روش های بالا، یافتن بخش آبی کنار پلاک و یافتن کل پلاک توسط تناسب طول و عرض پلاک، روش بهتری بود اما باز هم دقت کافی را نداشت زیرا تشخیص دادن رنگ آبی در بین رنگ های مختلف سخت بود. پس از مقادیری جستجو و کمک گرفتن از هوش مصنوعی، به فضای HSV رسیدیم:

در فضای رنگی HSV، هر رنگ به سه مؤلفه تقسیم می شود: Saturation (S)، Hue (H) و Value (V).

Hue زاویه ای در یک دایره رنگی است که نوع رنگ را مشخص می کند. به عنوان مثال، 0 درجه برای رنگ قرمز، 120 درجه برای سبز و 240 درجه برای آبی است. Saturation یا اشباع رنگ، میزان خلوص رنگ را نشان می دهد که بین 0 (خاکستری) تا 1 (رنگ خالص) متغیر است. در نهایت، Value به معنای روشنایی رنگ است که از 0 (کاملاً سیاه) تا 1 (کاملاً سفید) تغییر می کند.

در فضای HSV، تشخیص رنگ آبی بسیار راحت تر از فضای RGB است. در فضای RGB، برای تشخیص هر رنگ باید ترکیب سه رنگ اصلی (قرمز، سبز و آبی) به صورت همزمان بررسی شود که سخت است اما در فضای HSV همان Hue به تنهایی به ما می گوید که هر پیکسل متعلق به چه رنگی است. به عنوان مثال، برای پیدا کردن رنگ آبی تنها کافی است بازه ای از مقادیر Hue را بررسی کنیم که این رنگ را شامل می شود؛ که با اندکی جستجو مشخص شد بین 240 تا 300 درجه است.

- حالا همانطور که در تشخیص اعداد و حروف پلاک ها، اینکه تصویر را باینری کنیم به ما کمک میکرد، الان هم در تشخیص رنگ آبی میتوان از همان مفهوم باینری کردن استفاده کرد، یعنی برای هر پیکسل

بررسی میکنیم که آیا در محدوده ی منسوب به رنگ آبی قرار دارد یا خیر. اگر پیکسل در محدوده رنگ آبی باشد، به آن مقدار 1 (سفید) داده می‌شود و اگر نباشد، مقدار 0 (سیاه) دریافت می‌کند. نتیجه این کار یک تصویر باینری است که فقط بخش‌های آبی تصویر به صورت سفید (با مقدار 1) و بقیه تصویر به صورت سیاه (با مقدار 0) نمایش داده می‌شوند.

- اما استفاده از محدوده ی رنگ آبی باز هم آنقدر که باید، دقت نداشت، نهایتاً نکته ای که به ذهنمان رسید این است که در پلاک‌ها، همانطور که اعداد و حروف دارای رنگ مشکی خیلی خالصی هستند و زمینه هم رنگ سفید نسبتاً خالصی دارد، رنگ آبی کنار پلاک هم معمولاً هم خلوص بالایی دارد و هم نسبت به نور فلش و دیگر نور ها، دارای بازتاب نسبتاً زیاد و روشنایی زیادی است، پس برای مقدار های saturation و value نیز یک مینیمم تعریف کردیم که با حدس و آزمایش به مقادیر نهایی آنها رسیدیم و از آنها نیز در فرایند باینری کردن عکس بهره گرفتیم.

- در این میان تلاش کردیم این نکته که این بخش آبی کنار پلاک، دارای شکل مستطیلی است را هم در نظر بگیریم که در طول کد ها به آن اشاره میکنیم.

- در نهایت پس از شناسایی ناحیه آبی، با توجه به اینکه طول این ناحیه تقریباً یک نهم کل پلاک است، ابعاد کل پلاک را تخمین می‌زنیم. و در نهایت، پلاک را از تصویر اصلی برش داده، نمایش داده و ذخیره می‌کنیم.

حالا که روند یافت الگوریتم را فهمیدیم، مرحله به مرحله روی کد جلو می رویم:

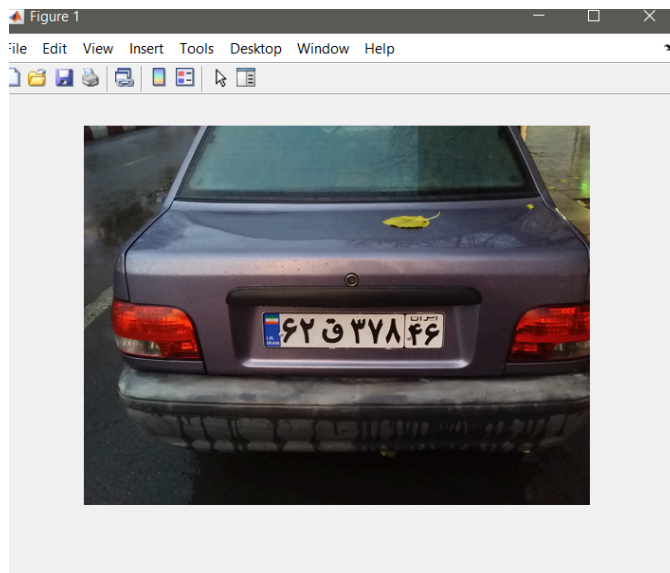
1. در مرحله ی اول همانند بخش های قبلی ابتدا تصویر را از کاربر گرفته و سپس آنرا به یک سایز نرمال رسانده تا اگر اندازه ی عکس خیلی بزرگ یا خیلی کوچک بود، قابل رویت و بررسی شود.

```
clc
close all;
clear;

%% Load the image
[file, path] = uigetfile({'*.jpg;*.png;*.jpeg'}, 'Select an image');
imagePath = fullfile(path, file);
img = imread(imagePath);

%% Resize the image
img = imresize(img, [400, NaN]);
figure, imshow(img);
```

مثلا:

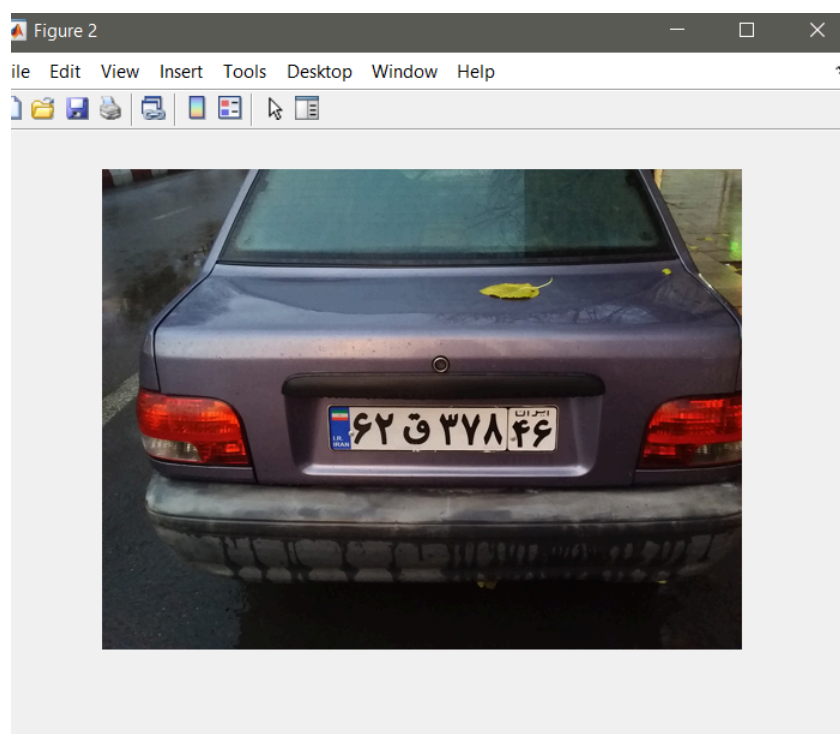


2. در این مرحله بنابر دلایلی که ذکر شد از جمله تشخیص راحتتر یک رنگ خاص مثل آبی روی یک جنس خاص مثل پلاک فلزی که رنگ هایش خلوص و روشنایی بالایی دارند، تصویر را بوسیله ی

تابع `rgb2hsv` به سیستم HSV می بریم و آنرا نمایش میدهیم(تفاوتی در نمایش آن بوجود نمی اید که محسوس باشد)

```
%% Convert the image to HSV color space (to detect blue more easily)
hsvImage = rgb2hsv(img);
figure, imshow(img);
```

مثلا:



3. حالا یکسری `threshold` که بنابر حدس و آزمایش روی پلاک ها و عکس های مختلف به آن ها رسیدیم را تعریف میکنیم برای Hue و Value و Saturation و سپس با توجه به آنها، به هر پیکسل لیبل 0 (به معنای نداشتن شرایط مورد نیاز و رنگ مشکي) و یا 1(بمعنای داشتن شرایط مورد نیاز و رنگ سفید) میزنیم .


```

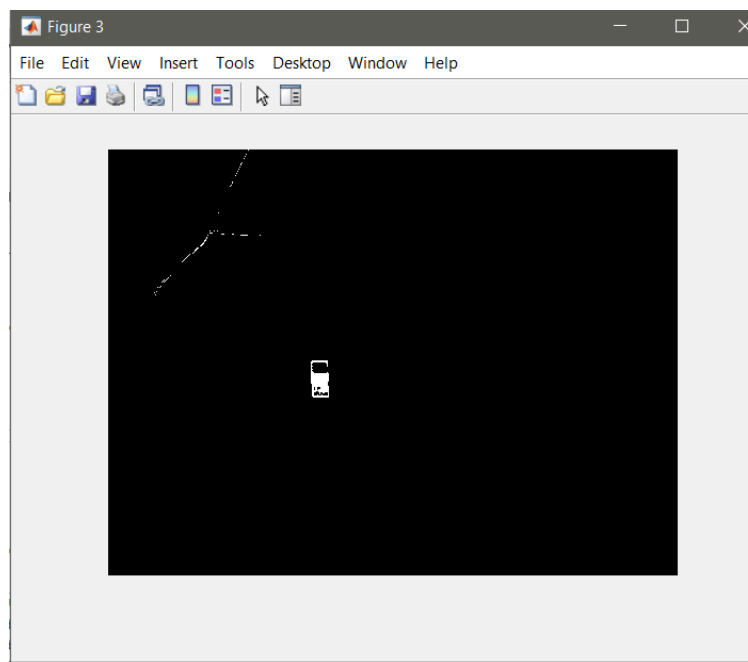
%% Define thresholds for the blue color in HSV
blueHueMin = 0.58; % Minimum hue for blue
blueHueMax = 0.75; % Maximum hue for blue
blueSatMin = 0.4; % Minimum saturation for blue
blueValMin = 0.2; % Minimum value for blue (brightness)

%% Create a binary image
binaryImg = (hsvImage(:,:,1) >= blueHueMin) & (hsvImage(:,:,1) <= blueHueMax) & ...
            (hsvImage(:,:,2) >= blueSatMin) & ...
            (hsvImage(:,:,3) >= blueValMin);

figure, imshow(binaryImg);

```

مثلا:

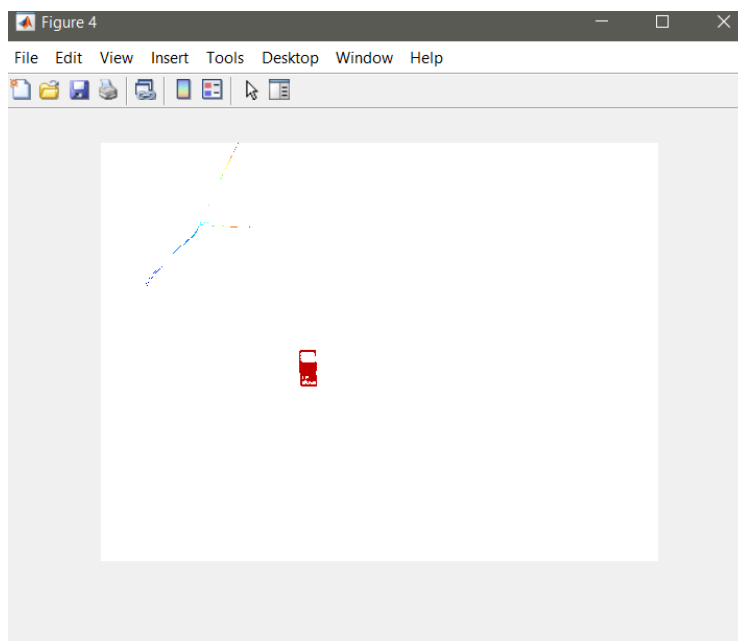


همانطور که حدس میزدیم، گوشه ی پلاک که رنگ آبی دارد، جزو بزرگترین محدوده هایی است که تمام ویژگی های مدنظر را داراست.

4. حالا جهت یافتن بزرگترین component ای که شرایط انتخاب شدن بعنوان گوشه ی پلاک را دارد، از تابع `bwlabel` استفاده کرده و کامپوننت های مختلف را لیبل میزنیم. روش نمایش اینگونه عکس های لیبل دار را هم در بخش های قبلی گفتیم.

```
%% Label connected components
[LabeledImg, num] = bwlabel(binaryImg);
figure, imshow(label2rgb(LabeledImg)); % Convert labels to colors for visualization
```

مثلا:



5. حالا برای یافتن بزرگترین کامپوننت کافیه یک حلقه for زده و هر بار پیکسل های مربوط به یکی از کامپوننت ها را از روی لیبل اش یافته و بوسیله ی کمترین و بزرگترین سطر و ستون هایی که دارای نقاطی مربوط به آن کامپوننت هستند، جهات طرف آن کامپوننت را بدست می اوریم و در نتیجه می توانیم طول و عرض مستطیل در بردارنده ی آن کامپوننت را هم بدست بیاوریم که بعدا در یافت ابعاد کل پلاک بدردمان میخورد

سپس با شمردن تعداد اعضای وکتور شماره سطر های نقاطی، در اصل به تعداد پیکسل های آن کامپوننت یا عبارتی مساحت آن کامپوننت میرسیم و میتوانیم به این صورت تا اتمام حلقه ی for ، بزرگترین کامپوننت آبی را بیابیم.

```

%% find the largest blue component
maxArea = 0;
blueRegionIdx = 0;
for k = 1:num
    % Find the pixel coordinates of the current component
    [rows, cols] = find(LabeledImg == k);

    % Calculate the bounding box (Bounding Box)
    minRow = min(rows);
    maxRow = max(rows);
    minCol = min(cols);
    maxCol = max(cols);

    % Compute the width and height of the bounding box
    blueWidth = maxCol - minCol + 1;
    blueHeight = maxRow - minRow + 1;

    % Calculate the area of the region (Area)
    area = length(rows); % Number of pixels corresponds to the area

    % Find the largest region (the blue part of the plate)
    if area > maxArea
        maxArea = area;
        blueRegionIdx = k;
        blueEdges = [minCol, minRow, blueWidth, blueHeight];
    end
end

```

6. در این بخش نهایتاً پلاک را می یابیم: با توجه به عرض بخش آبی که قبلاً یافته ایم و این تقریب که طول پلاک حدوداً 11 برابر عرض آن بخش آبی است، به طول پلاک میرسیم. از طرف دیگر مختصات ضلع های چپ، بالا و ارتفاع بخش آبی کنار پلاک، دقیقاً همانند کل پلاک است و گوشه ی راست را هم که از جمع مختصالی گوشه ی چپ بعلاوه ی طول پلاک که محاسبه کردیم قابل یافتن است.

پس رسماً تمام اطلاعات لازم برای دانستن محل پلاک را داریم، پس براحتی تصویر اصلی را برش زده و به پلاک میرسیم

```

%% Extract the edges of the plate
% Estimate the right boundary based on the fact that blue part is 1/11 of the plate
blueWidth = bluestatus(3); % Width of the blue region
plateWidth = blueWidth * 11; % The whole plate is approximately 11 times the blue region width

% Define the plate bounding box using the top, bottom, left, and calculated right boundary
plateX = round(bluestatus(1)); % The left boundary is the same as the blue region's left boundary
plateY = round(bluestatus(2)); % The top boundary is the same as the blue region's top boundary
plateHeight = round(bluestatus(4)); % Use the same height as the blue region

% Calculate the right boundary by adding the plate width to the left boundary
plateRightX = plateX + round(plateWidth);

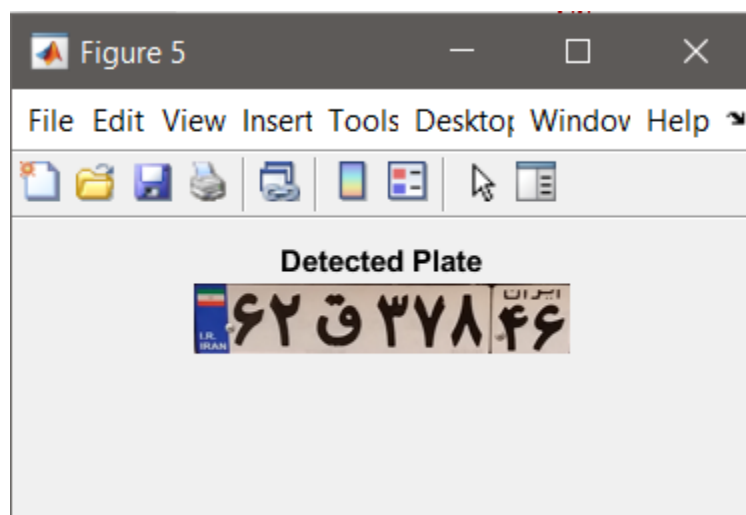
% Manually crop the plate using array indexing (rows and columns)
licensePlate = img(plateY:(plateY + plateHeight - 1), plateX:plateRightX, :); % Crop the region

% Step 10: Display the license plate
figure, imshow(licensePlate);
title('Detected Plate');

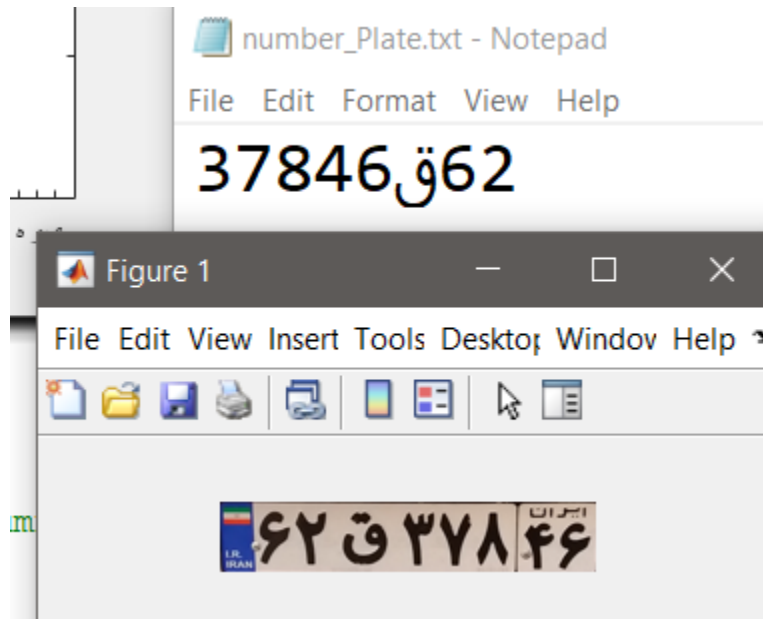
% Save the extracted license plate
imwrite(licensePlate, 'detected_plate.png');

disp(' plate detected and saved successfully.');
```

مثلا:



که دقیقا شبیه عکس هایی است که میتوان به فایل سوال دوم داده و متن را از آن استخراج کرد. اگر این کار را هم بکنیم داریم:



* همانطور که مشاهده میشود، با در نظر گرفتن جابه جایی کلمات فارسی در متن، دقیقاً به جواب درست رسیده و با استفاده از عکس جلوگیری، به اعداد و حروف پلاک رسیدیم.

