FCSE

Project Work
Digital Image Processing

License Plate Recognition with Python and OpenCV

Student: Nikola Toshev, 161022 **Mentor: PhD. Ivica Dimitrovski**

24 September, 2018

Abstract

Every day, computer vision has bigger role in the traffic, especially with invention of the self - driving cars. But computer vision was used on the roads way before their appearance. The basic jobs that do not require humans, such as license plate recognition were automated with computer vision systems. In this project, a multipurpose automatic license plate recognition software was built using the programming language Python and the open-source computer vision library OpenCV for the purposes of the Digital Image Processing course at the Faculty of Computer Science and Engineering - Skopje.

I. Introduction

Automatic license plate recognition (ALPR) takes an important role in many real-life applications e.g. automatic pay tolls, parking access control systems, average speed (between 2 cameras on a highway) and identifying arrest warrants. ALPR is a technology that uses optical character recognition (OCR) to automatically read license plate alphabets and numerals. Usually, stationary cameras are mounted on signs, street lights, buildings or highway overpasses. That way, moving vehicles are monitored, and software is used to detect a car plate pattern. The car license then is stored or queried (depending on the use case) in a database and certain action is taken. On the other hand, common use of mobile ALPR is recognizing parked cars in institutions such as university campuses. A vehicle with more cameras attached and a driver are enough to detect all the vehicles that aren't supposed to be parked in the car park.

I.I Goal

The aim in this project is to try to implement a software that recognizes and extracts license plate numbers and letters in a given image or video not dependant on the hardware devices (cameras or sensors) using the programming language Python and the computer vision library OpenCV. The idea is to create a prototype that can be used in various of applications and be slightly modified to fit ones needs, and not a standalone application. Once finished, the application should be able to extract the vehicle license characters in a given image or video and print them out at standard output with high accuracy.

II. Problem explanation

The problem itself is complex enough so we need to complete couple of tasks. Initially, we can divide the problem in two sub problems:

Extract the car plate/s in a given image (later we will generalize to videos too)
- Couple of techniques can be used to solve this mini-problem known as ROI recognition and extraction. Couple of solutions will be provided in the next section.

Recognize the digits in a given plate once it is separated from the image
- Machine learning classifier will be used to classify a given character once it has been extracted from the image. More classification algorithms will be discussed later, once we get to this step. Three images will be used to test the program and check the progress. In each one, the environment is different because the goal is to create an algorithm that generalizes well to different scenarios: different times of the day and different plate styles. The images are shown below.

Citroen_SK914VO

Hundai_SR370BM

Ford_M1637D

III. Solving the problem and discussing other possible solutions

1. Extracting the plate

The first step in solving the problem is detecting a license plate in a car image. Trying to think of possible solutions, one also needs to think of a technique that will manage to detect presence of a car plate in numerous scenarios. Only seeing the 3 testing images above, it is fair to mention that an algorithm working in one scenario may not be working in another one. So the problem comes down to object recognition problem, which is a very common computer vision problem. The next question we have to ask ourselves is: How is a license plate different than the rest objects within an image? Some general characteristics about the license plates:

- Plates have high contrast which is designed for humans to be able to read easily, and detecting high contrast is common problem in computer vision.
- Every country has limited license plate templates, so maybe using template matching where a license sample is the template. The idea is to find the almost identical part in the image such as the template (which in this case is a license plate).
- The location of a car plate is almost always towards the bottom and the middle of the car. Knowing that, the car can be detected (using the some other property), and the plate can be extracted regarding the car.
- The shape of almost every license plate in the world is a horizontal rectangle. Its aspect ratio is standard within one country.
- Combination of multiple techniques mention above

Trial and error. One of the best approaches in digital image processing and computer vision. One cannot know upfront whether a specific technique will work. Trying multiple ways to solve this I ended up using the last one which is simplest one to implement. Comments on using the other ones:

- Yes, plates have high contrast, but so does nature.

- Using template matching sounds like a good idea, but trying to implement it you realize that template matching is very sensitive. Using bigger threshold to neglect the characters may help that, but then the problem with rotating the plate stays unresolved.

- The idea to detect the vehicle's wheels first, for example, and then using their positions to detect the car plate location is probably the best one so far (if we know the angle of the camera). Because the idea is to build a multiple purpose demo application we cannot assume that.

High contrast image

Square license plate

1.1 Implementing the solution in Python and OpenCV

Every car plate is a horizontal rectangle with a specific aspect ratio. Or not? The rear license plate of the car above has the shape of a square, so using this technique may not be perfect, however it should work in most of the cases.

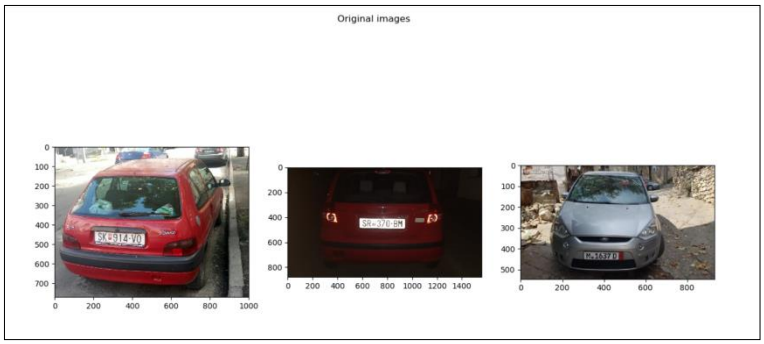
What was previously described, should be easily implemented with Python and the already mentioned computer vision library. First, let us create a Python module named shape_detector.py and import some useful dependencies for digital image processing.

```
1. #importing the dependencies
2. import numpy as np
3. from matplotlib import pyplot as plt
4. import cv2
```

These libraries will make the work easier, as each one of them is providing different functionalities. All three of them are widely used in computer vision.
- NumPy (Numerical Python) is a very commonly used python library, which adds support for multi-dimensional arrays and matrices, and collection of high-level super-fast mathematical functions to operate with these arrays.
- Matplotlib is a plotting library for Python and NumPy. It can be used to plot different types of data, with different types of graphs. In this work, it will be used to show the

results after some calculation or filter application.
- OpenCV is the key library that is going to be used, as it has all the functionalities needed to solve a given computer vision problem. In Python, we use cv2 to import this library. Let us first load and display the test images to see how easy it is to be done with OpenCV and Matplotlib. The result is shown just below the code.

```
#loading the images
car_1 = cv2.imread("cars/citron_sk914vo.jpg")
car_2 = cv2.imread("cars/hundai_sr370bm.jpg")
car_3 = cv2.imread("cars/ford_m1637d.jpg")

#create a new window to display the images
#display each testing image
fig1 = plt.figure(1)
fig1.suptitle("Original images")

plt.subplot(131)
plt.imshow(cv2.cvtColor(car_1, cv2.COLOR_BGR2RGB))

plt.subplot(132)
plt.imshow(cv2.cvtColor(car_2, cv2.COLOR_BGR2RGB))

plt.subplot(133)
plt.imshow(cv2.cvtColor(car_3, cv2.COLOR_BGR2RGB))

#finally show the result
plt.show()
```

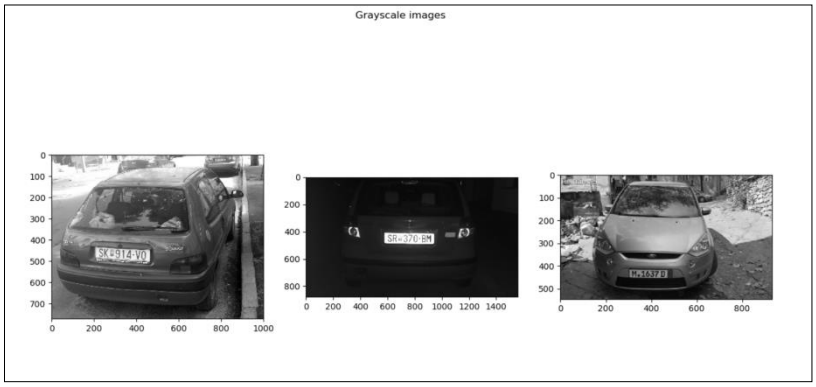


Result_1

In computer vision, most of the problems can be solved using grayscale images, as they are smaller in size, and most of the problems aren't dependant on color, but other features such as texture and shape.

```
#BGR to Gray scale conversion
gray_1 = cv2.cvtColor(car_1, cv2.COLOR_BGR2GRAY)
gray_2 = cv2.cvtColor(car_2, cv2.COLOR_BGR2GRAY)
gray_3 = cv2.cvtColor(car_3, cv2.COLOR_BGR2GRAY)

#display grayscale images
fig2 = plt.figure(2)
fig2.suptitle("Grayscale images")
plt.subplot(131)
plt.imshow(gray_1, cmap="gray")

```

```
plt.subplot(132)
plt.imshow(gray_2, cmap="gray")

plt.subplot(133)
plt.imshow(gray_3, cmap="gray")

#finally show the results
plt.show()
```



Result_2

It is expected that images and especially videos will have much noise. Noise removal filters are used to remove as much noise as possible. It is important to preserve the edges, because the idea is to detect rectangular shapes, and extract all of them. Picking the most suitable filter is very important, and through trial and error, it turned out that bilateral filter works very good (it smoothens the image, but the edges stay almost intact). After removing the noise, the next step is to detect the edges. There are numerous very accurate and efficient edge detecting algorithms already implemented in the OpenCV library. Having the denoised images now, if we apply Canny (available in OpenCV) we should expect pretty good results.

The way Canny edge detector works will be explained just below the result of the next commands:

```
#apply Canny edge detection to the filtered images
edges_1 = cv2.Canny(filtered_1,100,150)
edges_2 = cv2.Canny(filtered_2,100,150)
edges_3 = cv2.Canny(filtered_3,100,150)

#display edged images (some edges are lost if Matplotlib is used)
cv2.imshow("edged 1", edges_1)
cv2.imshow("edged 2", edges_2)
cv2.imshow("edged 3", edges_3)
```

Result_3

Seeing the results, it is noticeable that this approach is good (at least for the testing images). The license plate is a closed rectangular region which should be easily extracted. The Canny edge detector is an advanced edge detector, which goes through couple of steps:
1. Noise Reduction (even though we did some filtering previously, the built-in function does that again and it uses Gaussian Filter);
2. Applies Sobel kernel to the image horizontally and vertically to find derivatives and use that to find edges;
3. Non-maximum Suppression to make the edges thinner, because the previous steps may result in thick (more than 1px) edges.
4. The last step is thresholding: We pass 2 integer values as function arguments (0 - 255) named min_value and max_value. The function has its rules to decide which pixels are considered as an edge, and which ones are non-edges.

So far, so good. The next step is to detect the plate rectangle. Because it is a four-corner closed area, the problem shouldn't be tough. OpenCV contains a function named findContours which can be used to detect standalone objects in a binary image. Also, OpenCV has functions which are helpful to extract an object and approximate a rectangle. In the following code, the first function call is the one mentioned above (findContours) which requires three arguments:

-source image: which image do we search for contours;
-mode: maintain contour hierarchy or not;
-method: number of points that define the contour.

The remaining functions will be explained just under the following code chunk. From now on, only the first image will be used for testing, just to make things shorter.

```
#find contours based on the edges
_, cnts, _ = cv2.findContours(edges_1, cv2.RETR_LIST,
cv2.CHAIN_APPROX_SIMPLE)

#sort contours based on their area and keep the largest 40
cnts=sorted(cnts, key = cv2.contourArea, reverse = True)[:40]
rectangles = []
copy = car_1.copy()

for contour in cnts:
    perimeter = cv2.arcLength(contour, closed=True)
    approx = cv2.approxPolyDP(contour, 0.02 * perimeter, True)


```

```
#only if the detected contour has 4 corners retrieve the top
left x,y positions and the contour's width and height
    if len(approx) == 4:
        x,y,w,h = cv2.boundingRect(approx)


#the aspect ratio of a regular license is usually 4.5+1.5 margin
        if h > 0 and 3 < w/h < 6:
            rect = cv2.minAreaRect(approx)
            box = cv2.boxPoints(rect)
            rectangles.append(box)
            box = np.int0(approx)
            cv2.drawContours(copy, [box], 0, (0,255,0), 2)

cv2.imshow("After plate detection", copy)
```

Result_4

Once the contours are retrieved, using the sorted function they are sorted based on their area, and the largest 40 are selected (for efficiency reasons because some images may have many contours). For each contour, total number of corners is approximated using OpenCV approxPolyDP. Because our ROI has 4 corners we are only interested in such shapes. Compute each 4-corner contour's width and height and if the aspect ratio is approximately 4.5, then that is probably a plate. Some other objects might pass this condition, but they will be taken care of later. All the contours that satisfied the previous conditions are marked into a copy of the original image. Results are shown in figure named Result_4 and the matching contours are now stored in the variable rectangles.

Because the goal is to build a multi-purpose application, and some cameras might be placed at different angle (e.g. attached to a wall), some plates will be rotated for some degrees, and that will affect the selected ROI (won't be the desired one). Instead of the whole plate and nothing else, we might end up with rectangular shape that will contain other parts of the image. So if we show the selected region using the x and y position and the width and the height of the image, we will receive unexpected results:

Wrong section of the test image

Luckily, OpenCV has build-in functions to change the perspective of a given image. If the 4 points that define the rectangle are known, it is easy to change the perspective of the image. The following code does that:

```
plates = []

for pts in rectangles:
    rect = np.zeros((4, 2), dtype = "float32")
    s = pts.sum(axis = 1)
    #the top-
left corner has the smallest sum of x and y coordinates, and the bottom-
right has the biggest
    rect[0] = pts[np.argmin(s)]
    rect[2] = pts[np.argmax(s)]

    diff = np.diff(pts, axis = 1)
    #the top-
right corner has the smallest difference of x and y coordinates, and the bo
ttom-left has the biggest
    rect[1] = pts[np.argmin(diff)]
    rect[3] = pts[np.argmax(diff)]

    tl, tr, br, bl = rect

    # compute the width of the new image, which will be the
    # maximum distance between bottom-right and bottom-left
    # x-coordiates or the top-right and top-left x-coordinates
    widthA = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
    widthB = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
    maxWidth = max(int(widthA), int(widthB))

    # compute the height of the new image, which will be the
    # maximum distance between the top-right and bottom-right
    # y-coordinates or the top-left and bottom-left y-coordinates
    heightA = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
    heightB = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
    maxHeight = max(int(heightA), int(heightB))

    dst = np.array([
            [0, 0],
            [maxWidth - 1, 0],
            [maxWidth - 1, maxHeight - 1],
            [0, maxHeight - 1]], dtype = "float32")

    # compute the perspective transform matrix and then apply it
    M = cv2.getPerspectiveTransform(rect, dst)
    warped = cv2.warpPerspective(car_1, M, (maxWidth, maxHeight))
    warped = cv2.resize(warped, (180,40))
    plates.append(warped)
```

Noticeable difference, right? This step might require some additional computation but will be out of huge help when digit detection is performed.

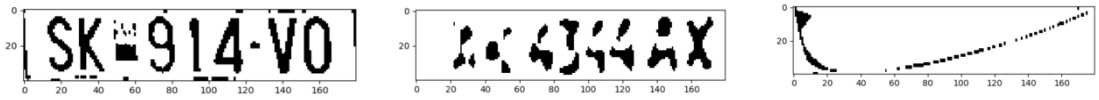
After changing the perspective

First, we sort the corners in the following order: top-left, top-right, bottom-right, bottom-left. Then calculate the maximum width and height, and compute the transform matrix. Apply to the original image and show the results. Resize to 180x40 (keeps the 4.5 ratio). Finally, store all the twisted images in an array: plates. This code can now be saved as function named detect_plates which takes one positional argument (the source image) and returns the warped images array.

2. Recognizing the digits/letters in an extracted plate

Once the plates are extracted, the next step is to get the application seeing what is written in there. In a function recognize_alphanumerics, which takes a plate array for processing. The idea is to load each plate, and try to separate each character from the others, again using the contours to do so. That means some filters are needed to remove the noise and the shadows, before it is searched for contours. So everything that is done, should be done to all the extracted "plates".

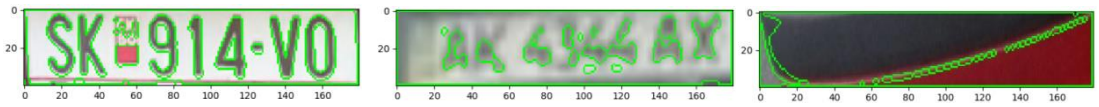
```
1. def recognize_alphanumerics(plates):
2.         for plate in plates:
3.
4. #apply shadow removing effect first
5.             rgb_img = cv2.split(plate)
6.
7.             result_img = []
8.
9.             for color in rgb_img:
10.                dilated_img = cv2.dilate(color, np.ones((3,3),np.uint8))
11.                bg_img = cv2.medianBlur(dilated_img,21)
12.                diff_img = 255 - cv2.absdiff(color, bg_img)
13.                norm_img = cv2.normalize(diff_img, diff_img, alpha=0, beta=
                   255, norm_type = cv2.NORM_MINMAX, dtype=cv2.CV_8UC1)
14.                result_img.append(diff_img)
15.
16.            removed_shadow = cv2.merge(result_img)
17.
18.            #threshold
19.            gray = cv2.cvtColor(removed_shadow, cv2.COLOR_BGR2GRAY)
20.            _, thresh = cv2.threshold(gray, 150, 255, cv2.THRESH_BINARY)
21.
22.            #closing
23.            kernel = np.array((5,5), np.uint8)
24.            closed = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel)
25.
26.            plt.figure(8)
27.            plt.imshow(closed, cmap="gray")
28.
29.            plt.show()
```



Result_5

Seeing the outputted images, it is easy to see which one responds to which contour. The second one isn't shown as expected, but it looks like that is because the test image has low resolution. The next step is to search for contours in the images and look after a successful way to define the letters and the numbers in a way that will separate them from the noise.

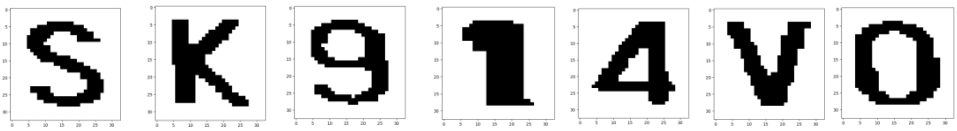
```
_, contours, _ = cv2.findContours(closed.copy(), cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
cnts = sorted(contours, key = cv2.contourArea, reverse=True)

cv2.drawContours(plate, cnts, -1, (0,255,0), 1)

plt.imshow(cv2.cvtColor(plate,cv2.COLOR_BGR2RGB))
plt.show()
```



Result_6

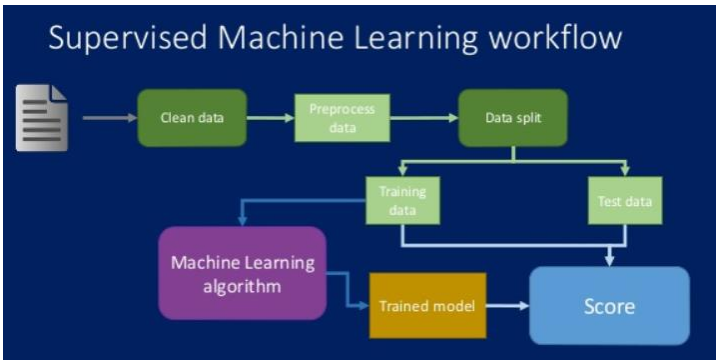
By adding some simple conditions, it should be easy to extract only the numbers, because they all have same height, and most of them have the same width. That means aspect ratio can be used to only detect the needed contours. Just to make sure the contours that we are aiming to select are more standard, some morphological operations and filters will be applied. Again, this was deduced through exploring different ways to find an accurate enough solution (trial and error manner).

```
for contour in cnts:
    #get x,y positions and contour's width and height
    x, y, w, h = cv2.boundingRect(contour)

    #select the contour region in the original image and resize it to 27x26 image
    roi = closed[y:y+h, x:x+w]
    roi = cv2.resize(roi,(27,27))
    _, roi = cv2.threshold(roi, 150,255,cv2.THRESH_BINARY)

    #dilate the image
    kernel = np.array((3,3),np.uint8)
    dilated_roi = cv2.dilate(roi,kernel,iterations = 1)

    #add white space around the contour
    bigger = np.vstack((np.ones([3,27],dtype = np.uint8)*255,roi))
    bigger = np.hstack((np.ones([30,3],dtype = np.uint8)*255,bigger))
    bigger = np.vstack((bigger, np.ones([3,30],dtype = np.uint8)*255))
    bigger = np.hstack((bigger,np.ones([33,3],dtype = np.uint8)*255))

    #convolution with 3x3 kernel
    dst = cv2.filter2D(bigger,-1,kernel)
    #detect characters that satisfy this loose condition and add them to array
    chars = []
    if 25 < h < 38 and 5 < w < 23:
        imgarr= []
         for row in bigger:
            for value in row:
                imgarr.append(value)
        chars.append(imgarr)

        #display the selected region
        plt.imshow(roi,cmap="gray")
        plt.show()
```



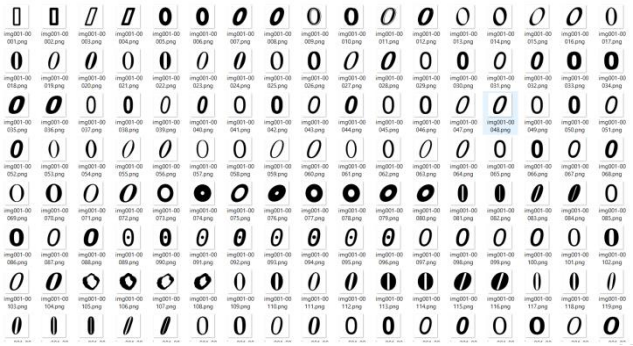
Result_7

Great! The only contours that were selected were the ones showed as Result_7. The blank space that was added may look unnecessary and weird, which may be true and is totally understandable. It will be clarified very soon. There aren't many choices to solve a character recognition problem. One can either add hand-written rules (which is ridiculous to do for 36 (26+10) characters, and even if we do, they will probably overlap and the recognition won't be accurate at all), or we can train some machine learning algorithm to solve the problem. Continuing from here with the obvious solution. Training any machine learning algorithm takes data. More reliable data - higher accuracy. To solve the problem we can try many algorithms, but for this one we will just use supervised learning, which workflow is explained in the following image:



Supervised Learning Workflow

Looking after a good free digits dataset wasn't easy because for some reason there isn't much on the Web. There are free datasets like the MNIST dataset that consists of digits only, but that wouldn't solve the problem with the letters. The best searching result that looked like would help was a set of font images. The folder consists of 62 subfolders (10 digits + 26 uppercase letters + 26 lowercase letters) and each one of those contains 1017 images:



Images example

Using this image set to train the classifier requires some data processing, because the images

are too big [128x128], they need to be resized, and their pixel values should be converted to numbers. Again, this can be done using Python. Seeing the images, it starts to make sense why the 3 white pixels were added from each side of each recognized number. Images can be converted to comma separated file, and use that file to train the ML classifier. Within the "Convert images to csv" module, images are loaded and resized to [33x33] with the OpenCV library, and stored in Pandas data frame. Pandas is a Python library for data manipulation and analysis and it can read, write to and create numerical tables. The data frame is stored as csv file with 36576 labeled observations and 1090 columns (33*33 pixels + label). The code that does that is showed below:

```
import cv2
import pandas as pd

cols = []
cols.append("character")
for i in range(1089):
    cols.append("px "+str(i))

df = pd.DataFrame(columns = cols, dtype='object')
for folder in range(1,10):
    path = 'C:\\Users\\Pc\\Desktop\\Fnt\\Sample00'+str(folder)+'\\'
    for i in range(1,10):
        img = path + 'img00'+str(folder)+'-0000' + str(i) + '.png'
        img = cv2.imread(img, 0)
        img = cv2.resize(img, (33,33))
        retr, img = cv2.threshold(img,150,255,cv2.THRESH_BINARY)
        img_data = []
        img_data.append(folder-1)
        for row in img:
            for value in row:
                img_data.append(value)

        df.loc[len(df)] = img_data


    for i in range(10,100):
        img = path + 'img00'+str(folder)+'-000'+ str(i) + '.png'
        img = cv2.imread(img, 0)
        img = cv2.resize(img, (33,33))
        retr, img = cv2.threshold(img,150,255,cv2.THRESH_BINARY)
        img_data = []
        img_data.append(folder-1)

        for row in img:
            for value in row:
                  img_data.append(value)
        df.loc[len(df)] = img_data

    for i in range(100,1000):
        img = path + 'img00'+str(folder)+ '-00'+ str(i) +'.png'
        img = cv2.imread(img, 0)
        img = cv2.resize(img, (33,33))
        retr, img = cv2.threshold(img,150,255,cv2.THRESH_BINARY)
        img_data = []
        img_data.append(folder-1)

        for row in img:
            for value in row:
                  img_data.append(value)
        df.loc[len(df)] = img_data

    for i in range(1000,1017):
        img = path + 'img00'+str(folder)+ '-0'+ str(i) +'.png'
        img = cv2.imread(img, 0)
        img = cv2.resize(img, (33,33))
        retr, img = cv2.threshold(img,150,255,cv2.THRESH_BINARY)
        img_data = []
```

```
        img_data.append(folder-1)

        for row in img:
            for value in row:
                  img_data.append(value)
        df.loc[len(df)] = img_data

for folder in range(10,37):
        path = 'C:\\Users\\Pc\\Desktop\\Fnt\\Sample0'+str(folder)+'\\'
        for i in range(1,10):
            img = path + 'img0'+str(folder)+'-0000' + str(i) + '.png'
            img = cv2.imread(img, 0)
            img = cv2.resize(img,(33,33))
            retr, img = cv2.threshold(img,150,255,cv2.THRESH_BINARY)
            img_data = []
            img_data.append(chr(ord('@')+folder-9))
            for row in img:
                for value in row:
                    img_data.append(value)

            df.loc[len(df)] = img_data

        for i in range(10,100):
            img = path + 'img0'+str(folder)+'-000'+ str(i) + '.png'
            img = cv2.imread(img, 0)
            img = cv2.resize(img, (33,33))
            retr, img = cv2.threshold(img,150,255,cv2.THRESH_BINARY)
            img_data = []
            img_data.append("Z")

            for row in img:
                for value in row:
                      img_data.append(value)
            df.loc[len(df)] = img_data

        for i in range(100,1000):
            img = path + 'img0'+str(folder)+ '-00'+ str(i) +'.png'
            img = cv2.imread(img, 0)
            img = cv2.resize(img, (33,33))
            retr, img = cv2.threshold(img,150,255,cv2.THRESH_BINARY)
            img_data = []
            img_data.append("Z")

            for row in img:
                for value in row:
                      img_data.append(value)
            df.loc[len(df)] = img_data

        for i in range(1000,1017):
            img = path + 'img0'+str(folder)+ '-0'+ str(i) +'.png'
            img = cv2.imread(img, 0)
            img = cv2.resize(img, (33,33))
            retr, img = cv2.threshold(img,150,255,cv2.THRESH_BINARY)
            img_data = []
            img_data.append("Z")

            for row in img:
                for value in row:
                      img_data.append(value)
            df.loc[len(df)] = img_data

df.to_csv("characters.csv",mode='a', sep=',', encoding='utf-8',index=False)
print("done")
```

The csv file is now ready to be used to train the classifier. Because the images were thresholded, the csv file contains two values for the pixels: 0 and 255. In a module "Create classification model.py", the data is read, and with the help of the machine learning specialized library "scikit learn" Logistic Regression model is created, trained on 90% of the data, and validated (10% of the images). Measuring the accuracy, the LR model had 97% percent accuracy for the validation set, which is great precision. The trained classifier is

saved as "LR characters classifier" pickle object. It can now be loaded and used in the main module to make predictions on a new observations. The following code does the previously explained job:

```
1.  import pandas as pd
2.  from sklearn.model_selection import train_test_split
3.  from sklearn.metrics import accuracy_score
4.  from sklearn.linear_model import LogisticRegression
5.  import pickle
6.
7.  #read the csv file
8.  df = pd.read_csv("characters.csv", sep = ",", dtype="unicode")
9.
10. #separate the dependant and independant columns
11. X_chars = df.iloc[:,1:]
12. y_chars = df.iloc[:,0]
13.
14. #split the dataset into training and test set
15. X_train_chars, y_train_chars, X_test_chars, y_test_chars =
16. train_test_split(X_chars, y_chars, stratify=y_chars, test_size=0.1)
17.
18. #create logistic regression classifier
19. clf = LogisticRegression(dual=False, tol=0.02, solver='liblinear', multi_class='ovr
    ')
20.
21. #train the classifier
22. clf.fit(X_train_chars, y_train_chars)
23.
24. #predit the testing values
25. y_pred_chars = clf.predict(X_test_chars)
26. print(accuracy_score(y_test_chars,y_pred_chars))
27.
28. with open('LR characters classifier.pickle', 'wb') as handle:
29.         pickle.dump(clf, handle)
```

3. Recognizing the digits/letters in an extracted plate

If all the previously written modules are put together in a "main.py", there is an already testable application. So, firstly the extract_plate function is used to detect an existing plate, and then recognize_alphanumerics is used to identify the letters and the digits. The main and the testing result is shown below:

```
1.  from license_recognition import extract_plate, recognize_alphanumerics
2.  import pickle
3.  import cv2
4.
5.  if __name__ == "__main__":
6.      with open("LR characters classifier.pickle", mode = 'rb') as handle:
7.          clf = pickle.load(handle)
8.
9.      plates = extract_plate(cv2.imread("cars\citr_1.jpg"))
10.     recognize_alphanumerics(plates, clf)
```

Result_8

The application caught the license plate twice, which resulted in 2 predictions:
1. SK PMQ VU;
2. SK P1Q VO
and the results aren't as good as they were expected

to be. Let's use more images to test the app.

Program output:

```
['M' '1' '6' '3' '7' 'D']
['M' '1' '0' '3' '7' 'D']
>>>
```

Program output

```
['S' 'R' '3' '7' '0' 'B' 'M']
['S' 'R' '3' '7' '0' 'B' 'M']
>>>
```

Program output:

```
['K' 'U' '3' '2' '3' 'L' 'A' 'D']
['K' 'U' '3' '2' '3' 'L' 'A' 'D']
>>>
```

Program output:

```
['E' '0' '5' '6' '5' 'K' 'C']
>>>
```

Program output:

```
Couldn't find plate
>>>
```

Program output:

```
['G' 'U' 'B' '7' '5' '3' 'A' 'C']
['G' 'V' 'B' '7' '5' '3' 'A' 'C']
>>>
```

It turns out that the letter recognition isn't that bad. There are some reasonable errors: 8 and B, 4 and L, U and V, 6 and O... And some not so obvious, such as 3 and 1, 4 and Q, 1 and M... The letter prediction is good, but there is high error when it comes to digits. It is just frustrating seeing digits conflicting with letters. That is when you start thinking about improving the classification model. After trying different algorithms such as Support Vector Machines, Decision Tree, and Random Forest, again the results weren't good. Apparently there is something wrong with the data. There are many images, but they are very different compared to the numbers used in plates, because **ML without good data is useless.** To demonstrate that, I have created a new dataset by hand that contained observations of only the digit. With the help of 40 detected shapes using the extract_plate function and the detected digits using the recognize_alphanumerics around 130 digits data was saved as .csv file, just like previously. This dataset had around 12 +-2 observations per digit compared to 1017 using the previous image collection. The idea now, is to train 2 classifiers: 1 for digits, and 1 for letters, the first one using the newly created dataset, and the other one using only the letters in the previously created file. This approach should improve classifying both the digits and the letters.

Not to repeat things, the 2 models are trained just like the previous one and named "LR digit classifier" and "LR letter classifier". The recognize_alphanumerics will now take 3 arguments: the plates, digit classifier and letter classifier. Switching to two classifiers has one big downside: we need to know the exact plate standard that we are going to be identifying, because the plates in different countries have different character arrangement. In this project, we are going to optimize this application to work for the Macedonian plates which have two standards: LL-DDD-LL and LL-DDDD-LL where L stands for letter and D stands for digit.

The code below was added at the end of the recognize_alphanumerics module to complete the application:

```
#the characters were added in a chars array
letters, digits = [], []

#LL-DDD-LL
if len(chars) == 7:
    letters = [0,1,5,6]
    letters = [chars[i] for i in letters]
    digits = [2,3,4]
    digits = [chars[i] for i in digits]
#LL-DDDD-LL
elif len(chars) == 8:
    letters = [0,1,6,7]
    letters = [chars[i] for i in letters]
    digits = [2,3,4,5]
    digits = [chars[i] for i in digits]

#make probabilistic predictions
if letters and digits != []:

    #digits
    digs = digit_clf.predict_proba(digits)
    for item in digs:
        maximum = max(item)
        index_of_maximum = np.where(item == maximum)
        print(index_of_maximum[0][0], " | confident:", maximum)

    #letters
    lets = letter_clf.predict_proba(letters)
    for item in lets:
        maximum = max(item)
        index_of_maximum = np.where(item == maximum)
        print(chr(ord('@')+index_of_maximum[0][0]+1), " | confident:", maximum)
print("------------------------------------")
```

The improved application should now be tested on new images. It is very important not to use some of the images that were used to create the digit training set, because these observations are familiar to the ML classifier.


```
7  | confident: 0.6044520622491181
0  | confident: 0.97887508920303
0  | confident: 0.9737929030384948
S  | confident: 0.9532034578428326
K  | confident: 0.9845932998473443
M  | confident: 0.4449543396536186
T  | confident: 0.9275932883693481
------------------------------------
------------------------------------
>>>
```


```
6   | confident: 0.984323238574568
2   | confident: 0.9628755594594771
1   | confident: 0.9881771967622589
4   | confident: 0.9969928146939623
S   | confident: 0.8235447761186331
K   | confident: 0.9890119123811787
A   | confident: 0.9894656463734443
T   | confident: 0.6133336689909218
------------------------------------
------------------------------------
>>>
```


```
5   | confident: 0.9947218655368876
4   | confident: 0.9966051780444523
2   | confident: 0.9740753401131391
7   | confident: 0.9956972346453232
B   | confident: 0.9555413257958771
T   | confident: 0.9407475959543016
A   | confident: 0.9633139708122939
B   | confident: 0.986397495979496
------------------------------------
```


```
1   | confident: 0.9931173894446247
7   | confident: 0.9920984672803157
0   | confident: 0.9920451264658712
7   | confident: 0.9957583791847161
S   | confident: 0.920239435809304
K   | confident: 0.8653546191205129
A   | confident: 0.9583922103275918
K   | confident: 0.7469274390412389
```


```
------------------------------------
1   | confident: 0.75677916428646
1   | confident: 0.8135323998656672
6   | confident: 0.972578377824063
4   | confident: 0.9929212088166538
K   | confident: 0.9641142701277141
U   | confident: 0.9889810756222827
D   | confident: 0.5475746359196986
E   | confident: 0.9347062402267217
------------------------------------
1   | confident: 0.9717480800225574
1   | confident: 0.919484269766435
6   | confident: 0.9834540574141991
4   | confident: 0.9941137240195527
K   | confident: 0.9865865428711151
U   | confident: 0.9944649242673439
C   | confident: 0.4441790592518844
E   | confident: 0.9855245291337962
```

Testing the application

After the improvement, it starts to feel like better results were accomplished. The digit classifying in images is almost perfect, and the letter classifier is evidently improved. The added confidence is very helpful, because seeing the images and the results above, the lower the confidence - the lower the precision. That means: these kind of cases can be handled with human intervention. Now, it is time to see how does this demo app perform if the input is a video instead of image. Videos are much noisier than images, and worse results are expected .

[video here]

IV. Conclusion

The final plate recognition module is not yet ready for production. There are cases in which it is failing to first: detect a plate existence (mostly in videos due to low resolution or even in images if there is something unusual: shadow, weird positioning and rotation, etc.), and second: recognize the digits, even though the plate was detected, the chars in it are blurred, unclear, or the whole plate is just noisy and the characters cannot be separated right. The small dataset for digit recognition does the job for images, but the classifier isn't highly accurate (or at least not very confident about the predictions) when it comes to noisy videos.

Object detection and recognition is not a trivial task even when using high-level libraries like OpenCV, since each record is different than all the other ones. Only because of the variety of possible scenarios, there isn't a best approach that will work in every single one. That is the only obstacle that is stopping this application from being completed without much more work. Probably, the best approach to address an ALPR problem is: once you have the practical scenario in which the software is going to be used, and the settings are familiar, try to implement a practical solution, e.g. pay tolls, parking lot access control system, etc. It is much easier to work on the problem if you know the location in which the license will appear. To create an app that never misses a plate, and predicts almost 100%, it is clear that another technique should be used (e.g. the approach to detect the rectangular shape wasn't the best out of the multiple choices provided at the beginning of this text).

This project (in my opinion) wasn't complete failure. The model can predict license plates in clear images pretty good, and can be improved even more with minor touch-ups (see Future Work section).

V. Future Work

If more work was to be done on this project, the main focus should be directed towards two sub-problems: developing better ROI detector, and gather more data to train a Deep Neural Network. The first one requires rewriting code from scratch, and the second one doesn't require much coding, but still is not easy to accomplish (needs some searching and maybe even gathering data if appropriate data cannot be found).

An alternative approach is to detect the existence of a license plate, is to know its approximate position. Once the position area is established, the high contrast can be used to only keep the region of interest of the image. This way, it is guaranteed that a plate will almost certainly be extracted, which results in robust system. When a video is used to detect license plates, the execution is slower than expected. The previously explained approach might work faster than the current one.

When small datasets are available (which is in this case), using machine learning algorithms works well, because they do not need much observations to learn a specific rule. The main issue in the numerical dataset is the lack of noisy examples. When used in videos,

because the quality is lower, the classifier is "confused" (that is probably why it fails to predict right in some video frames, or is less confident about the predictions). If a dataset which contains 100 observations of each digit (including noisy ones), this demo application will hardly fail to predict right.
What was done with the digits, can be done with the letters too. It only requires more time and nothing technical.
If a big dataset (lots of observations) is created, instead of using traditional ML algorithm, a DNN (deep neural net) can be trained and used to classify the images. In general, deep learning almost always manages to outperform machine learning. The only downside is that it needs much more data to learn from, requires more processing power, and takes more time to complete the training phase. That being said, switching to DL in the future can also be considered.
Last suggestions for future improvements: one: change the usage of aspect ratio to detect plates using the bounding box's height and width because this fails in some cases (use the ones after changing the perspective) and two: find a more robust way to detect the characters (this method hasn't yet failed while testing, but seems very insecure).

VI. References

[https://stackoverflow.com/]
[https://docs.opencv.org/3.4/]
[http://www.numpy.org/]
[https://matplotlib.org/]
[http://answers.opencv.org/questions/]
[https://pythonprogramming.net/]
[https://opencv-python-tutroals.readthedocs.io/en/latest/index.html]
[https://www.pyimagesearch.com/]