Project Work for Computer Vision

Davide Giordano
davide.giordano6@studio.unibo.it

chkrr00k
chkrr00k@cock.li

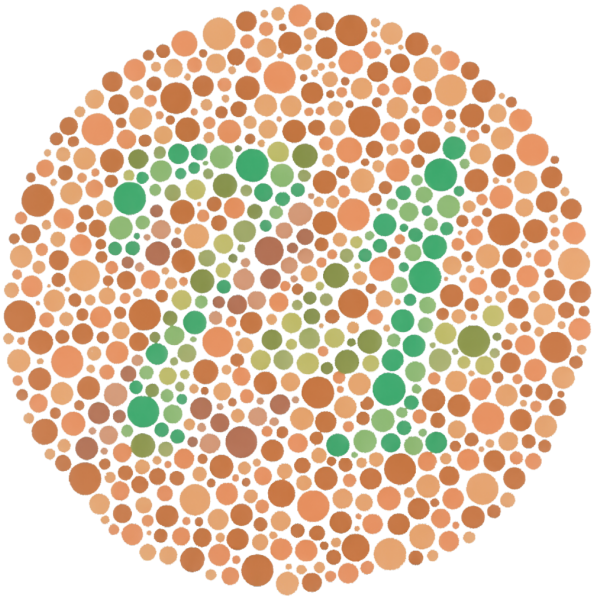
Abstract

The project consists in reading and solving Ishihara plates in an automatically way. This process is basically divided in two points, the first is dividing the main colors in the plate and the second is actually reading the number.

1 The Ishihara plates

Ishihara plates are composed by two (or more) main colors slightly shifted in the color plane so to be a little intersected and harder to tell apart by colorblind/color impaired people. Some plates even have multiple numbers inside in a way so people with normal vision see a number while some specific kind of colorblind people see another number to tell apart the cases.[5]
As side note it's important to say that not all tables contain numbers and not all tables not containing numbers are actually void tables because they could have "patterns" inside so to allow children to take the test as well without having the bias of not knowing properly the numbers.

Figure 1: Ishihara plate with two number inside: 74 for normal vision and 21 for red-green colorblindness



2 Color quantization

The "official" method according to the *OpenCV*[4] site to perform color quantization seems to be the use of *k-means* algorithms.

2.1 Otsu's method

A first idea could seem to apply an Otsu-like binarization but this doesn't make sense for multiple reasons. First of all Otsu's method is a one dimensional method as it works only on grey levels, this leads to the

conclusion to run Otsu for each coordinate in the (i_r, i_g, i_b) space. But this also doesn't lead to the searched result. Applying Otsu for each coordinate just means to have a vector $I = (i_r, i_g, i_b)$ meaning that

$$p(i) = \left(\frac{h(i_r)}{N} \quad \frac{h(i_g)}{N} \quad \frac{h(i_b)}{N}\right)$$

If we keep applying this method the result is that also μ and σ^2 are defined as a three dimensional vector. This doesn't make sense because it means that each coordinate is not correlated to the other meaning that the division on each axis will be independent to the other thus dividing only where the per color inter-group variance is minimized but not where the *global color inter-group variance* is minimized. This means that the meaning of the Otsu method result will not be the one needed to solve this problem.

Another important aspect to note is that if we want to correctly apply Otsu's method in more than one dimension while keeping a valid meaning and a correct result the appropriated method is called *Linear discriminant analysis* or in the specific a discrete *Fisher's linear discriminant*. These methods are equivalent of Otsu's but in multidimensional cases. It's also provable that they are equivalent to an optimal *K-means* search[6] and so having all problems *K-means* have. These problems will be discussed in the next section.

2.2 K-means

The *K-means* algorithm, as noted before, allows you to divide a set of elements in a given number of classes by using clusterizations. An important aspect of the *K-means* algorithm is that you need to know already in how many parts you have to divide your given set (i.e. you need to know beforehand how many colors are there in our application case) This has some limitations as the number of color present in the actual plate has to be given to the algorithm before the execution meaning that the number or the structure of a specific plate has to be known beforehand. This is obviously illogical as to know the correct K you need to know which K is the one that solves the problem meaning that you not only already ran the algorithm for that plate, but that after you ran it you saw that the correct result was shown.
Since we can't understand the correct K to obtain a feasible solution our only option would be to "cycle" over all possible K until a feasible solution has been reached. An ulterior consideration is that if you cycle on the K to find the feasible one, to know that is feasible means that we processed all results through an OCR algorithm. This is not only slow but also very much not optimal.

2.3 Local maxima analysis

A valid alternative is to study how the colors are distributed in the plane. The Ishihara plates are structured to have various main colors which are the ones that forms either the number or the background. A strategy could be to select only those color within a mask. This strategy is also nonsensical because it assumes to know the color to select. To solve this some transformation can be made: for starters it's best to ditch the BGR scheme as it's not the human vision model but just a model directed to machines and representation in a 3D plane. A better model is HSV that has a third channel (V) that describes the brightness of the given pixel. That brightness is responsible for most of the color variations in the Ishihara plates and as consequence it can be set to max value to simplify the quantization. At this point we are basically looking at a 2D plane with all the base color.

From RGB to HSV To have a proper conversion from RGB to HSV the following algorithm is applied:
Having value $X_{max} = V := \max(R, G, B)$, $X_{min} := \min(R, G, B) = V - C$ meaning that chroma $C := X_{max} - X_{min} = 2(V - L)$ and lightness $L := \frac{X_{max} - X_{min}}{2} = V - \frac{C}{2}$. This means that the common hue will be

$$H := \begin{cases} 0, & \text{if } C = 0 \\ 60° \cdot \left(0 + \frac{G-B}{C}\right), & \text{if } V = R \\ 60° \cdot \left(2 + \frac{B-R}{C}\right), & \text{if } V = G \\ 60° \cdot \left(4 + \frac{R-G}{C}\right), & \text{if } V = B \end{cases}$$

And the saturation:

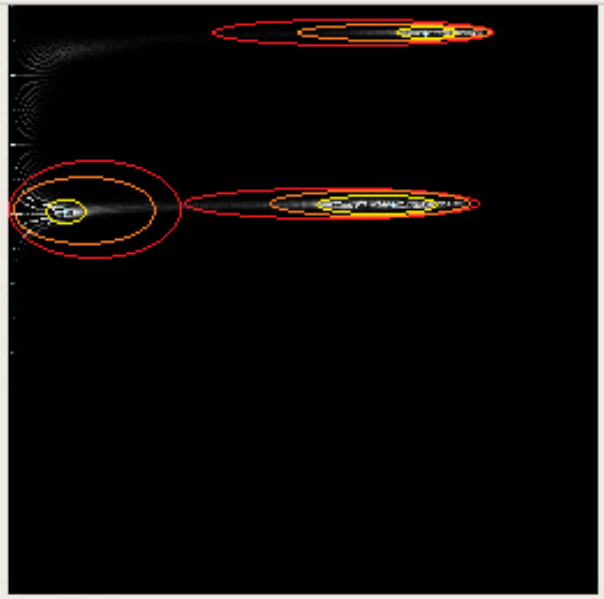
$$S_V := \begin{cases} 0, & \text{if } V = 0 \\ \frac{C}{V}, & \text{otherwise} \end{cases}$$

At this point we quantize all the present colors in *buckets* so every pixel in the image will be in a specific color bucket that bares the given color. This is very easily done but has some computational cost (technically $O(n)$):

```
buckets = dictionary()
for pixel in image:
    if pixel in buckets:
        buckets[pixel].append(coordinates(pixel))
    else:
        buckets[pixel] := [coordinates(pixel)]
```

After the creations of such buckets we know exactly how many colors there are and since the Ishihara plates have main color with other slightly colors distributed around them we can assume the main colors will be *local maxima* in a 3D graph with $(H, S, len(bucket(H, S)))$ as axis.

Figure 2: Color density map to locate the local maxima as main colors



A naive approach to find these local maxima is to define an ϵ as a neighbourhood of the proposed maximum so it'll be researched in that window. A problem that occurs is that in the (H, S) space the distances aren't really euclidean because on one axis there is more color change than the other. An obvious solution to this is to use a *Mahalanobis distance* to emphasises only the important axis to avoid taking the wrong colors. The current proposed matrix is:

$$\begin{pmatrix} \frac{1}{6} & 0 \\ 0 & 1 \end{pmatrix}$$

Given these assumption the local maxima algorithm proceeds to sort all buckets in order of popularity (i.e. to have the one with most pixels first) with supposed complexity of $O(n\,log(n))$. From that sorting it's assumed that the data is now in order and if we run through the buckets now we'll have the points from the most common to the least one.
The next phase will be to select the most popular pixels in a given neighbourhood by using the following naive algorithm (which is around the $O(n)$ if the distance is well set):

```
local_maxima = list(bucket[0]) //the first bucked with be a local
                               //maxima since it's the global one too
for color in buckets[1:]:
   for current_maxima in local_maxima:
       if distance(color, current_maxima) < epsilon:
           compressed := True
           //set the current_maxima as a possible compressor of the given color,
```

```
            //meaning that the color is actually in the neighbourood of the current maxima
    if not compressed:
        local_maxima.append(color)
    else:
        #set as compressor of color the current compressor
 #append to each compressor set the actual maxima
```

This not only gives us the local maxima but also allows us to quantize the color in a pixel list that can be used later to generate a mask.

2.4 Comparing the two methods

Given an image with 77556 pixels (so to speak a 276x281) a *k-means* algorithm set to find 3 labels (i.e. the background white, the main color and the background color) takes 0.237 seconds to complete, but for the given images the result not very satisfactory (3) as the number is very hard to tell apart and to process for an OCR. The first occurence where the number is actually distinguishable is when $K = 6$, an arbitrary number that works only for that plate (5). With a local maxima analysis, however, we can see how the color division are preserved and categorized properly with the given color (6)

Figure 3: *k-means* with $K = 3$

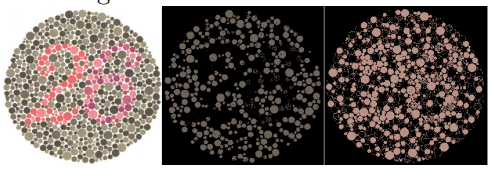


Figure 4: *k-means* with $K = 4$

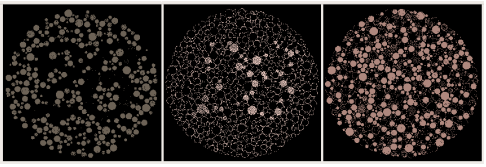


Figure 5: *k-means* with $K = 6$

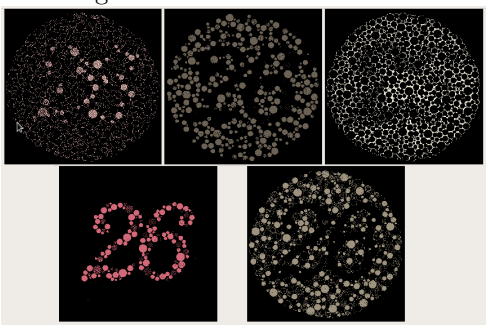
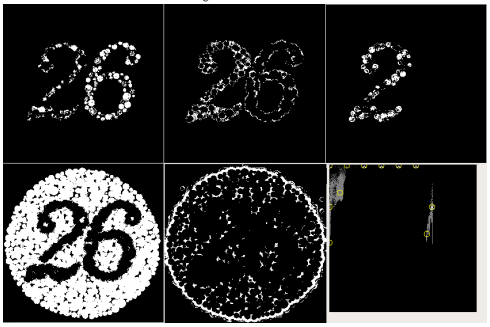


Figure 6: Proposed local maxima analysis with $\epsilon = 100$ and default distance matrix



2.4.1 Efficiency

As time efficiency we have to take into account the dimension of the given image but also the accuracy of the result.

Method	Phase	Time (s)	Size of Image	Result
k-means ($K = 3$)	-	**0.237**	(276x281)	Insufficient
k-means ($K = 6$)	-	**0.472**	(276x281)	Acceptable
Local maxima	Bucket	0.498	(276x281)	-
-	Maxima search	0.395	-	-
-	-	**1.188**	-	Acceptable
k-means ($K = 3$)	-	**2.321**	(1080x1000)	Insufficient
k-means ($K = 10$)	-	**5.464**	(1080x1000)	Acceptable
Local maxima	Bucket	3.658	(1080x1000)	-
-	Maxima search	5.953	-	-
-	-	**12.969**	-	Acceptable

From this data we can deduce that the proposed local maxima method is less fast, but is more efficient *result-wise* as it doesn't need a given parameter if not the given ϵ and the *Mahalanobis* matrix which are however given and generic for such problem while *k-means* needs a different parameter for each plate. Using different code some results where the local maxima was actually **faster** than the *k-means* was also yielded.

3 The best image

The previous method leaves us with some binarized images, one for each maximum found. The task at this point is to select the "best" one, so the one that actually holds the number in itself. If we have this image we can proceed with the detection of the number in it. Selecting the best image is not trivial, as the number in it could be hidden by noise or just be very hard to detect. This has been solved by applying a sorting algorithm over some parameters for each image after a filtering round to eliminate some unwanted masks.

3.1 The bound assumptions

Some assumptions have been applied to reach this "algorithm", the first one is that we are operating with images with only one number in the center of the Ishihara plate, even if it must be noticed that this algorithm gives a very high result score even for double number plates. Another assumption is that we know the font of the number in the plate. Both assumptions allow us to create out own generator of Ishihara plates to have an extensive testing work-frame. Our generator currently takes in a number (or just a supported glyph, really) and procedurally generates a plate around it. The current generation algorithm is not efficient but it's outside this report's scope to implement an efficient *CSP*-based Ishihara plate generator.
The current generator operates within these color bounds:

Foreground	0xF9BB82
	0xEBA170
	0xFCCD84
Background	0x9CA594
	0xACB4A5
	0xA3A260
	0xBBB9640
	0xD7DAAA
	0xD1D6AF

A careful analysis of an Ishihara Plate will lead to the result that the number is actually in a subset area of the whole plate. This allows us to research specifically in that area, this area has been referenced in the "Negative area ratio" in the plate. This area is given by the user and it's roughly $h/5 < y < h * 4/5$, $w/8 < x < w * 7/8$ with w, h the width and the height of a given mask, this will be referenced as the number bounding area.

3.2 The filtering

These parameters have been chosen after a careful analysis of the attributes:

Number of contours	Let c found contours, $c > \textbf{avg(\# contours)} * \epsilon$	$\epsilon = 0.2$
Dominance of white pixels	Let w white pixels, t all pixels, $\textbf{avg(w)} * \epsilon_u > w * t > \textbf{avg(w)} * \epsilon_l$	$\epsilon_u = 1.4, \epsilon_l = 0.3$
Median area	Let ma the median contour area, $a > \textbf{avg(ma)} * \epsilon$	$\epsilon = 0.6$
"Negative" area ratio	Let p the percentage of contours in a given bound, $A_{\textbf{in}}$ the average area in the bound, $A_{\textbf{out}}$ the average area outside the bound, $A_{\textbf{in}} * p > A_{\textbf{out}} * (1 - p) + \epsilon$	$\epsilon = 0$

When applying this filters a *caveat* must be followed: every time a filter filters **all** the masks, such filter must be voided and a rollback to the previous state must be applied as it's obvious that we are in a special case where such parameters aren't in the assumptions of the problem. It must be noted that in our tests no rollback has ever happened.

3.2.1 Meaning of the filters

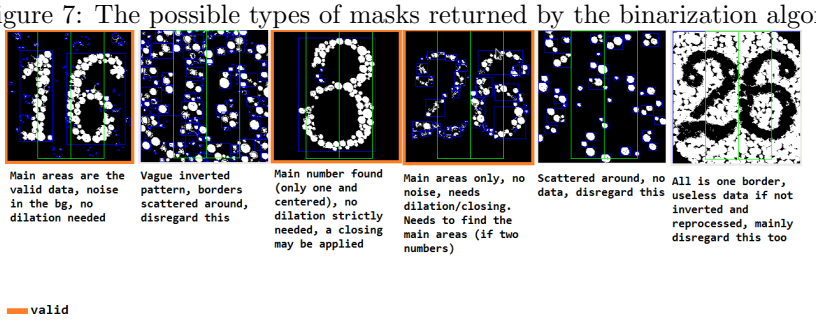
The number of contours allows us to filter out those images that have too few contours if compared to the others, this means a mask that have little or no pixels, or the opposite case, a too big contour as the case of a completely white mask.

The dominance white pixel is the percentage of white pixels in relations to the background this also allows us to discard those images that have a too big mask. This allows us to remove masks with white background as they represents the extraction of the background color in the plate.

The median area allows us to filter out those masks where the contours are too small so their area is subsequently small, and so removing all those masks generated by either artifact noises or just improper binarization.

The negative area ratio, maybe the most important of the set, is the one that allows us to take only the masks that have a considerable majority of pixels and contours in the bound area and so we know that they have probably a number in them and it's not just background presence in the bounding area.

Figure 7: The possible types of masks returned by the binarization algorithm



3.3 The ranking

After applying all these filtering, the next step is to sort the remaining masks in descended order with a sorting algorithm based on the "In bound average area ratio" percentage. This allows us to select the image that have the highest number of contours inside the bound area, meaning it's more likely to have a number inside.

3.4 Cleaning the image

Once the best image is selected a set of cleaning procedures is applied to have a better final image. The cleaning is operated in two steps:

3.4.1 Preliminary cleaning

In this step the image is both dilated and subjected to a median blurring filter. In order a median blur is applied to the selected mask and then, after a selection of a proper elliptic structuring element which scales from the size of the image to maintain a scalability and modularity approach, a morophological closing operation is applied.

3.4.2 A neighbourhood filter

This operator performs a cleaning by adding pixels if certain conditions are met in the image. This operator is applied for each pixel and it follows this algorithm [1]:

```
for all pixels(x, y) in image:
    if size(neighbours(3)) > 9:
        pixel(x, y) := white
```

This process allows us to "fill" those spaces that might have been left behind form the binarization due to noise.
It should be noted that this is not an LSI due to the fact that the superimposition principle is not valid: Let's assume to have two matricies A and B s.t.

$$A = \begin{vmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{vmatrix}, B = \begin{vmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{vmatrix}$$

their sum would be

$$A + B = \begin{vmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{vmatrix}$$

. This means that if we call the defined filter $neigh(I, neighbourSize, point, threshold)$ on a neighbour set of 1 on the $(1, 1)$ element of $A + B$ we'll have

$$neigh\,(A + B, 1, (1, 1), 4) = \begin{vmatrix} 0 & \textcircled{1} & 0 \\ 0 & 0 & \textcircled{1} \\ \textcircled{1} & 0 & \textcircled{1} \end{vmatrix} = 1$$

But if we apply it to A and B

$$neigh\,(A, 1, (1, 1), 4) = \begin{vmatrix} 0 & \textcircled{1} & 0 \\ 0 & 0 & \textcircled{1} \\ 0 & 0 & 0 \end{vmatrix} = 0$$

and

$$neigh\,(B, 1, (1, 1), 4) = \begin{vmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \textcircled{1} & 0 & \textcircled{1} \end{vmatrix} = 0$$

Which means that

$$neigh(A, 1, (1, 1), 4) + neigh(B, 1, (1, 1), 4) \neq neigh(A + B, 1, (1, 1), 4)$$

proving that *additivity* is not respected

After these passages, the image is then passed to an OCR implementation.

4 Reading the characters (An OCR implementation)

To research the best OCR implementation some considerations have to be made. For starters there is never a best way to perform such operation because of the volubility of the system. This paper explores some "naive" approaches like an area-based ocr, which assumes both the shape, size and the font are known by the program, an ever more naive approach done with template matching (which by definition *can't* be successful) and some more smarter approaches based on Machine Learning.

4.1 Area-based

This is perhaps the most naive of the approaches as the program need to know the size of the letters and the font *a priori*. This is very much not scalable but in theory could yield high results due to the fact that it's a solution tailored around this specific problem.

4.1.1 The implementation

As expected the implementation is very naive and all it does is to generate all possible glyphs in the known fonts and group them under the same glyph label G. It will then apply two operations to the given mask image I:

$$M = \max_{g \in G} \{\text{Count}(I \wedge g) - \text{Count}(I \wedge \neg g)\}$$

This means that the algorithm will select the label having the most area overlapping a given glyph subtracted the part outside the area. The pseudocode is:

```
for g in G:
    current = count(I and g) - count(I and not g)
    if current > count_max:
        glyph_max = g
        count_max = current
```

Obviously the actual implementation is a bit more refined but the idea behind it it's this one.

4.1.2 The results

A good accuracy has been obtained by using this method, but a big *caveat* is that the glyphs have to be known and so the font. This reduce the scalability of the system and it's flexibility. Another big problem is that for some particular numbers (i.e. 5 and 6) the area difference resulted to be too small to be noted by the system and the OCR often mistook 5 for 6 during testing phases.

4.2 SSD and SAD

A different approach is to start from a template-like object and to calculate the "distance" of the character we have to read from the given archetypes. In this case SAD is defined as the Sum of Absolute Differences and SSD as Sum of Squared Differences. SAD is by far faster due to the absence of the square. They are both used in an envirnoment where the intensity does not change due to the fact they are not intensity invarant like ZNCC and NCC are, in our case this will not happen both because our starting images were already digitally created but also because they were binarized during the blob analysis phase.
In the project an implementation of ZNCC and NCC also exists and has also yielded very good results with high accuracies. (As an *addendum* it must be noted that while SAD and SSD finds their better match in their minimum, ZNCC adn NCC have their best match at their maximum). The idea behind both SAD and SSD is to have a sliding window as big as the template image, then on every pixel of both the image and the template it'll be computed a substraction to measure the various distance between the two.

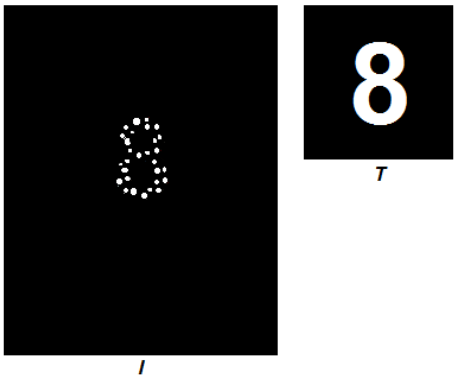
$$SSD(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \left(I(i+m, j+n) - T(m,n)\right)^2$$

$$SAD(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} |I(i+m, j+n) - T(m,n)|$$

4.2.1 The implementation

The implementation is based on template matching, but the concept behind it is to calculate the distance from various archetypes representing all possible numbers in the image. The closer one (either with L2 or L1 distance) will be the number present on the image. As said the implemented algorithm uses a sliding window on the base image I having size of the template image T, the template will be generated by creating a different template for each character that can be in the table using possibly the same font. It follows that the template will be compared to the binarized image, so the template will also have a binarized form.

Figure 8: Example of image and generated template



4.2.2 The results

Since this is not technically exact template matching our result differs from the standard *SSD vs SAD* comparisons. In our case SAD yielded better results than SSD this is due to the fact that we are not looking for a template but the lower distance from a picture this means we are looking for the most similar, but not the exact one, thus SAD returned better results and was more suitable for this application. What follows is that ZNCC and NCC will also be affected by that in different measures but still yield good results. Tests of those had as resulting accuracy of respectively 99% with both algorithms so guessing incorrectly only in 2 cases over 150 plates.

4.3 SIFT

This approaches uses an exact template matching approach. Since the template matching theory tells us that the purpose is to find a template T in an image I we already know that this experiment is *technically incorrect*. It could however yield outstanding results if it was applied for a printed character recognition system. This approach has been added in this paper to serve as a baseline and to show how, even if these are technically printed character, this problem can't be solved with a basic recognition system and some consideration have to be applied.

4.3.1 The implementation

This implementation mainly uses the *OpenCV* function for SIFT[7].

After an image to read has been given to the algorithm the function will detect all keypoints and descriptors to confront them with a set of precalculated and labelled descriptors one for each character to detect. After that an homography will be searched in those keypoints using a RANSAC algorithm. The predetermined set is calculated from a representation of the searched characters as if they were never converted into Ishihara plate form. As it may appear evident this is not how to perform a proper SIFT or template matching search as the template might only be something that is actually present in the image and not something whose shape just resemble it. This is due to the fact that the descriptors and keypoints are based on edge detection and in this particular cases there are no similar edges because of the "bubbly noise" introduced by the *ishiharization* of the numbers.

Another important point is that the template images as well as the mask aren't at all optimal for the SIFT algorithm also because they are already binarized and so having a lot less of details will result in also a smaller number of keypoints which as consequence led to the need to reduce the keypoints found threshold which just raised the number of false positive without having any positive connotations.

4.3.2 The results

The result of this test has been (as expected) abysmal. Not only it's hard to find a single homography - since often the homography gets "spoofed" by noise and improper binarization - but the inability of the algorithm to find a unique solution has made this algorithm very inaccurate and capable of hitting a 0% accuracy in particularly unlucky cases.

Another important factor to consider is that in case more than one homography is found the system will perform a random selection of the homographies found to decide which character is in the plate. This

was made for the simple reason that if the system saw homographies it already knows that the two are equivalent thus selecting one over the others makes no sense. The idea to also select the one with more similar descriptors also doesn't hold because of the noise of the image which would just means similar edges but not in the correct positions and so having no meaning.

As stated before this was just a test to prove the actual incorrectness of this method in this particular case. It's important to notice that if the character weren't in the Ishihara plate form but just printed character this method would have been the best way to find printed OCR since template matching is exactly the way to go if an exact subimage is searched in an image.

4.4 SVM

This has been the first machine learning approach that was attempted with the goal of recognize characters. It uses the actual *OpenCV* libraries[9] and is based on Support-Vector Machines. This method is a statistical learning system that uses kernel functions to separate various classes of objects. This method aims to find the smallest convex hull containing all elements in a specific class, then it'll calculate some hyperplane dividing the space and containing all hulls while also being the farthest from each other. A subset of the points in the hulls is enough to characterize the hulls, those points are called *support vectors*. The basic concept is in fact to give categorized examples for each label, then the system will calculate some definition for each class to get the mapping via a labelling function. An important note is to remember that what we called *kernel* here has no relation ship with any kernel from computer vision theory as it's just in *function kernel*. This method, as all other machine learning method that has been tried is based on supervisioned approach.

4.4.1 The implementation

Since SVM uses a series of different parameters such as C, γ, the kernel type a simple script to find the best combination of those has been run. The result of that script while using *GridSearchCV*[11] were the following:

Kernel type	C	γ	Precision	Recall
rbf	1	0.001	0.014	0.1
rbf	1	0.0001	0.014	0.1
rbf	10	0.001	0.014	0.1
rbf	10	0.0001	0.014	0.1
rbf	100	0.001	0.014	0.1
rbf	100	0.0001	0.014	0.1
rbf	1000	0.001	0.014	0.1
rbf	1000	0.0001	0.014	0.1
linear	1	-	0.913	0.94
linear	10	-	0.913	0.94
linear	100	-	0.913	0.94
linear	1000	-	0.913	0.94

From those result it has been decided to use a linear kernel with $C = 1$ as it was the optimal choice.

Beyond the OpenCV implementation Since the run test file was actually implemented using the sklearn SVM function and not the OpenCV one it has been decided to implement the OCR using the sklearn provided function too. This verified both the correctness of the parameter and yielded result compatible with the OpenCV implementation.

4.4.2 The results

This method had high accuracy in both of the implementation, being able to correctly identify all given plates with very small trainsets (e.g. trainsets of size 10, so only one different labelled plate for each given glyph).

4.5 kNN

This method is also based on machine learning and has yielded outstanding results where in particular situations gave an accuracy of 100%. This OCR is implemented using k-nearest neighbours. The idea

behind this method is to give k examples for each class, if it's in classification mode when given an element to classify the algorithm will return *the class which is nearest to the given element.*
As all the machine learning it needs a training set. To get it we generated a various size traing set composed by equal number of Ishihara plates having as common feature only the number inside for each class.

4.5.1 The implementation

This implementation also uses the *OpenCV* kNN classes. As expected it needs to calculate a train set, which will be given by randomly generating known plates with known numbers inside thus providing a valid labelling. This implementation is very standard and no particular changes have been made since the theory of kNN OCR is already well settled[8]. Like for the SVM implementation this method need a parameter k and to find the fittest value some testing have been run:

k	Precision	Recall
1	0.485	0.57
2	0.406	0.51
3	0.472	0.54
4	0.488	0.55
5	0.535	0.59
6	0.529	0.59
7	0.473	0.55
8	0.397	0.48

From those result it has been decided to use $k = 5$.

4.5.2 The results

The results of this method have been outstanding and, in some occasions hitting accuracies of 100%. In some early testing this algorithm correctly identified 310 numbers on 330 proposed plates, for an accuracy of 94%.

4.6 Gaussian Naive Bayes

This method is statistical based having its basis on the Bayes theorem. In this method is assumed that every attribute of a class is independent from the others, from this assumption it takes the name of *naive*. As the methods above it also needs a train set to get its statistical values.

$$P(H|E) = \frac{P(E|H) \cdot P(H)}{P(E)}$$

Where H is the hypothesis and E the evidences based on that hypothesis. The hypothesis is the class, so a label, and the evidences are the values of the element to classify.

4.6.1 The implementation

This method was found in the sklearn libraries and has been adapted to solve the OCR problem. Since it needs a smoothing parameter a set of test has been run to find the optimal one:

smoothing	Precision	Recall
10	0.74	0.80
1	0.99	0.99
0.1	0.817	0.88
0.01	0.66	0.75
$1e-5$	0.616	0.72

From those result it has been decided to use 1.

4.6.2 The results

This method also had very good result even with very small train sets.

5 Conclusions

After running a comparisons of all these methods the results show that the best one was *SVM* but other machine learing methods such as kNN, GNB and scikit's SVM were also very good. These tests also shows how the area based one and SAD also yield good results even if some issues in particular cases with some characters. As expected the most elaborated methods are the ones that got better results, but it must be stated that they are also slower so there is a valid consideration about trade off between accuracy and speed when performing any operation.
The tests were performed by running a script that run the program with various dataset size to be sure of its actual accuracy. Then after a dataset was generated various test set were tried to avoid particular cases.

Method	Average	# Train set	# Test set	Total avg
GNB	0.98	10	50	-
-	0.92	20	50	-
-	0.98	100	50	-
-	-	-	-	0.96
kNN	0.96	10	50	-
-	0.96	20	50	-
-	0.98	100	50	-
-	-	-	-	0.97
SVM	0.98	10	50	-
-	1.00	20	50	-
-	0.98	100	50	-
-	-	-	-	0.99
SkSVM	0.94	10	50	-
-	0.98	20	50	-
-	0.98	100	50	-
-	-	-	-	0.97
SIFT	0.04	N.D.	50	-
-	0.04	N.D.	50	-
-	0.08	N.D.	50	-
-	-	-	-	0.06
Area	0.86	N.D.	50	-
-	0.82	N.D.	50	-
-	0.80	N.D.	50	-
-	-	-	-	0.82
SAD	0.86	N.D.	50	-
-	0.84	N.D.	50	-
-	0.90	N.D.	50	-
-	-	-	-	0.87
SSD	0.26	N.D.	50	-
-	0.22	N.D.	50	-
-	0.22	N.D.	50	-
-	-	-	-	0.23
NCC	0.96	N.D.	50	-
-	1.00	N.D.	50	-
-	1.00	N.D.	50	-
-	-	-	-	0.99
ZNCC	1.00	N.D.	50	-
-	0.98	N.D.	50	-
-	0.98	N.D.	50	-
-	-	-	-	0.99

6 Additional informations

The actual implementation is done in python3 and employs some strategies and patterns that such language allows us to have. Everything was made with modularity, scalability and testing in mind with an almost agile-like approach for the development.

The software is currently hosted on *GitHub*[2] for a better team synchronization.

6.1 Implementation details

The program itself presents as a python3 executable file, it has different parameters and settings that can be passed to it. From the help of the script[3]:

```
-h, --help              Displays this help
-k, --ocr <type>        Select the type of ocr
-t, --train             Trains the ocr
-s, --size <int>        Selects the size of the train set [default = 2]
-l, --load <file>       Loads the trained file
-d, --dump <file>       Saves the trained data
-v, --verbose           Verbose prints
--debug                 Enables debug features
-a, --accuracy <int>    Calculates the accuracy
-c, --char <char>       Specify the char to test
--silent                Produce no output
-j <json file>          Select ocr modules file
-p, --show              Show the images and the internal elaboration passages
```

This allows the user to have a better customization and an easier testing. Follows an example of how to run a simple testing with the kNN OCR, training on a 10 sized train set and checking the accuracy with 30 generated images for three consecutive times:

```
$ python3 test.py -k knn -d data_set -t -s 10 -a 30 --verbose
$ python3 test.py -k knn -l data_set -a 30 --verbose
$ python3 test.py -k knn -l data_set -a 30 --verbose
```

6.1.1 How to run the program

Run to detect the number 0 in a generated ishihara plate with area matching

```
$ python3 test.py -k area -t --char 0 --verbose --show
```

Run to calculate accuracy with 10 images in a generated ishihara plate with area matching

```
$ python3 test.py -k area -t --accuracy 10 --verbose --show
```

The available $-k$ parameter are:

- none - the default test one, will not return a result
- sift - the sift ocr implementation will be run, only -t is supported and -l, -d may not be used.
- knn - the knn ocr will be run, it will require either a data set passed via -l or to generate one via -t.
- svm - the svm ocr will be run, it will require either a data set passed via -l or to generate one via -t.
- sksvm - the sksvm ocr will be run, it will require either a data set passed via -l or to generate one via -t.
- gnb - the gnb ocr will be run, it will require either a data set passed via -l or to generate one via -t.
- area - the area ocr will be run, only -t is supported and -l, -d may not be used.
- sad - the sad ocr will be run, only -t is supported and -l, -d may not be used.
- ssd - the ssd ocr will be run, only -t is supported and -l, -d may not be used.

6.1.2 Accuracy

The accuracy is measured by, after getting a (trained) model, asking it to read a number n of plates and then applying the following formula:

$$A_{ccuracy} = \frac{n_{hits}}{T_{otal}}$$

Why not recall We used accuracy and not recall due to the fact that all passed images are expected to have a number in them, thus the concept of recall as $R_{ecall} = \frac{tp}{tp+fn}$ becomes effectively $R_{ecall} = \frac{tp}{tp} = 1$ (for $fn = 0$ as there can't be false negative in this set just incorrect classifications) and so an useless metric.

6.2 Software Engineering

The code is organized to be the most scalable possible to help with the development and extension of the program. It's divided in various modules each having a different meaning.

6.2.1 Modules

By using python's concept of modules a greater code separation and deduplication has been achieved. Here's a list of the most important present modules:

- generator - contains the base function to generates an Ishihara plate given a glyph (or a set of glyphs).
- renderer - contains the function to generate a glyph in a void canvas. It's used by the generator module to render the plate.
- extract - contains the base function to extract the optimal mask containing the number from a given RGB image.
- categorize - implements the functions to bucketize the pixels in a given image. Used as first step by the extract module.
- maxima - contains the base function to calculate a local maxima with the bucked method in a given image (or 2D array). Used by the extract module after the bucketization.
- selector - contains the base function to selects the best image in a given set. Used by the extract module after the masks have been created.
- visual - implements functions that take care of generate images or printing shapes. Mostly used by the extract module.
- neighbour - implements the cleaning function to apply before the OCR reading.
- utils - contains various utility functions used in the whole program.
- ocr - contains the OCR interface.
- various modules such as knn, sift, svm - implements the OCR interface and the actual OCR algorithms.

6.2.2 OCR interface

Python doesn't itself support any interface-like object oriented style, however it's possible to use the Abstract Base Class (abc) module to emulate that. By using that module it was possible to create an abstract *OCR* class to derivate the other implementations of specific aforementioned OCR implementation.

```
import abc
class OCR(abc.OCR):
    @abc.abstractmethod
    def read(self, img, **params):
        pass
```

```
    @staticmethod
    def get_train_set(size, glyphs=GLYPHS, verbose=False):
        pass
```

As seen there are two main functions, one is the *read(...)* which is the function that, when passed an image, will attempt to read the character present in that image. The other one is a static method used to retrieve a generic train set to train a Machine Learning or Neural Network system. It has to be noted that not all implementations are forced to use the *get_train_set(...)*. The actual training of the eventual network must be done in the *__init__(self, dump=None, load=None, train_set=None, verbose=None)* constructor which have to be compliant with that signature and, if not implementing some functionality, may raise a *ValueError*.

The expected implementation of the interface should:

- *dump* - if it's a path it will save the trainset there.
- *load* - if it's a valid path file it will load the trainset from there.
- *train_set* - if it's a positive integer it will create a trainset of that specified size (or if no size is neaded it will just generate one) and proceed with the training of the network/init the inner workings.
- *verbose* - as expected this just enables debug and verbose printings.

The implementation is also supposed to implement the *with ... as ...* python pattern interfaces for an easier implementation of the program in future steps. Since the main uses a system based on the following pattern:

```
with ocr_class(train_set=train, dump=dump, load=load, verbose=verbose) as o:
    r = o.read(img)
```

With:

- *ocr_class* - the dynamic OCR class implementation selected by the input parameters.
- The parameters of the class respecting the OCR class interface.

This allows the system to have a better and more lightweight implementation with a very scalable and modular approach.

Reflection To allow even more scalability it's possible to link *any* class to the program that implements the OCR interface. This is done by using *reflection* on the module loader via the python's *importlib*[10] module. Any known class can be linked by adding to the *module.json* file a new module and class name using the following pattern:

```
{
    "ocr": {
        "<-k parameter name>": {"cls": "<class name>", "mdl": "<module name>"}
    },
    "default": "<default ocr>"
}
```

After parsed and converted to a python dictionary that code will be used by the importlib to load the class. In the specific what happens is:

```
m = importlib.import_module(settings["k"]["mdl"]) # dynamic module import
ocr_class = getattr(m, settings["k"]["cls"]) # given the module m,
                                             # get the requested class
```

References

Ayatullah Faruk Mollah, S. B., Nabamita Majumder, & Nasipu, M. (2011). Design of an optical character recognition system forcamera-based handheld devicess. *IJCSI International Journal of Computer Science Issues*. Retrieved from https://arxiv.org/ftp/arxiv/papers/1109/1109.3317.pdf

chkrr00k, d. (2021a). *Ishivision.* Retrieved from https://github.com/chkrr00k/IshiVision

chkrr00k, d. (2021b). *Ishivision's site.* Retrieved from https://chkrr00k.github.io/ishivision
Gary Bradski, A. K. (2008). *Learning opencv.* O'Reilly Media, Inc.
Ishihara, S. (1972). Design of an optical character recognition system forcamera-based handheld devicess. *Kanehara Shuppan.* Retrieved from http://www.dfisica.ubi.pt/~hgil/p.v.2/Ishihara/Ishihara.24.Plate.TEST.Book.pdf
Liu, D. (2009). Otsu method and k-means. *Ninth International Conference on Hybrid Intelligent Systems IEEE.*
OpenCV. (n.d.-a). *Introduction to sift (scale-invariant feature transform).* Retrieved from https://docs.opencv.org/master/da/df5/tutorial_py_sift_intro.html
OpenCV. (n.d.-b). *Ocr of hand-written data using knn.* Retrieved from https://docs.opencv.org/3.4/d8/d4b/tutorial_py_knn_opencv.html
OpenCV. (n.d.-c). *Ocr of hand-written data using svm.* Retrieved from https://docs.opencv.org/master/dd/d3b/tutorial_py_svm_opencv.html
Python. (n.d.). *importlib — the implementation of import.* Retrieved from https://docs.python.org/3/library/importlib.html
scikit learn. (n.d.). *Gridsearchcv.* Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html