PIVA - Finger detection

Daniel Molinuevo and Antoni Jubés May 2020

Introduction

There is a huge variety of applications where skin and finger detection could be applied. From fingertip detection to enable human-machine communication, to real time deaf-language translation. In the following report, we discuss an approach to detect the amount of fingers shown in an image by means of a two-step algorithm: a first step based on skin detection and a second one that, from the results of the previous, that is able to count how many regions corresponding to fingers are present in the image. Therefore, the problems presents two different challenges: being able to define what characterizes skin regions and discovering common features regarding finger shapes and positions in the hand.

Since the size of the database we have is relatively small (60 images for training and 46 for validation), we present an approach that does not use standard machine learning or deep learning. Some statistical analysis is used, as well as simple image processing techniques. The results are quite promising and some of the proposed ideas could still be improved in order to apply them to different application.

Color spaces

In order to process images that are not black and white, we need strategies that allow us to represent the value of the 'color' in each pixel. This is usually done by means of a 3-dimensional representation, which classically corresponds to the amount of colors red, blue and green, which we will discuss in the following section.

Depending on the application this representation might not be the most appropriate, therefore we will discuss many others, specially those that could help us perform the task at hand.

RGB

This was one of the original choices, since it has a relation with how the human eyes actually perceive color: The human eye has three kinds of photoreceptor cells (called 'cones') and each one of them focuses on perceiving the amount of light around a different wavelength (this idea can be seen in the following image). These 3 wavelengths correspond to what we call 'red', 'blue' and 'green.

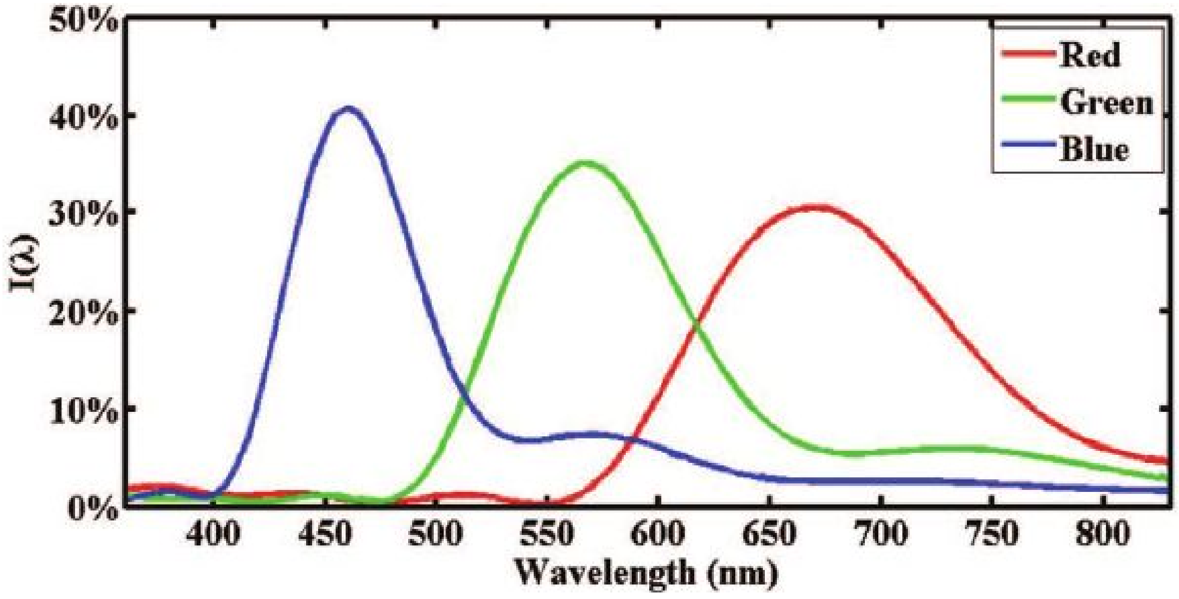

Figure 1: Color Spectrum.

This is one of the main reasons that a 3-dimensional representation where each dimensions represents red (R), blue (B) and green (G) respectively is widely used. The values go from 0 to 1, but they might be expressed as an integer between 0 and 255 (the size of a byte) as that is the most usual representation.

It has the problem that each dimension, although we know it represents the amount of each color, doesn't have an intuitive meaning in the sense that it if we are told, for example, that the R dimension has a value of 135, it is hard to grasp a meaning in terms of what's happening in the image or what is present there. That is why, for many concrete applications, we need to switch to other representations.

Sometimes, in RGB or some of the other color representation, a gamma correction is applied, which maps the original values to some new by means of a power-law function. This allows to take into account the non-linearity of our color perception that can be seen in the previous figure.

YUV

Initially used to re-code RGB for transmission efficiency in TV broadcasting. It consists of a linear transformation of the original RGB space into a luminance component (Y) and two chrominance components (U, V).

The luminance is computed as a linear combination of the R, G and B values, each of them ponderated by a coefficient so that the three of them add up to one:

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

We can see that if R, G and B are normalized between 0 and 1, Y will be as well. This version corresponds to the one used in Intel IPP and it is an example on how this could be computed, different variations can be found.

One would think that 3 chrominance components are needed (one for each color), but we would found that we are mapping a 3-dimensional space into a 4-dimensional space, therefore one of the chrominance components would be a linear combination of the other two plus the luminance, thus resulting in a redundancy of information.

In Intel IPP, the two chrominance components are computed as follows:

$$U = 0.492 \cdot (B - Y)$$
$$V = 0.877 \cdot (R - Y)$$

It can be seen that they are associated to the red and blue components.

The main advantage of this system is that the luminance component can be encoded with more bandwidth, since it is where most of the important information is present (our eye is more sensible to it), whilst the chrominance components can be transmitted with less bandwidth. This idea can be seen in Figure 2, where luminance and chrominance have been splitted.

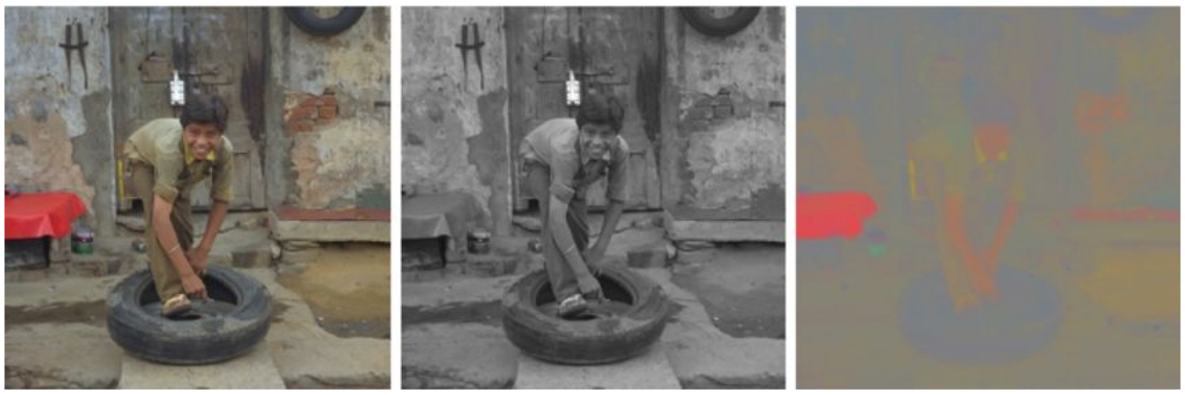

Figure 2: Luminance-chrominance example.

YCbCr (YCC) and YPbPr

These approaches are very similar to the YUV representation, as all of them are Luminance-Chrominance type of models, they search first for the luminance and then they both define the two dimensions related to the chroma of the image. The difference between these two models and the YUV one is on the definition of the chrominance dimensions, which will be presented later.

These two models are practically the same one, the only difference being that the YPbPr is the one used in analog devices, and the YCbCr is the one used in digital devices.

Now we define 3 constants KR,KG,KB and as we did in the YUV model we calculate Y:

$$Y = KR \cdot R + KG \cdot G + KB \cdot B$$

And we calculate Pb and Pr, which corresponds to the chrominance dimensions before applying the rescaling:

$$PB = \frac{1}{2} \cdot \frac{B' - Y}{1 - KB}$$
$$PR = \frac{1}{2} \cdot \frac{R' - Y}{1 - KR}$$

And this two last values are usually rescaled so that Cb and Cr vary in the range [16,240]. The value for the constants of the luminance are usually the same in all the luminance-chrominance type of models, and so, using this information we could get the value for the constants PB and PR too.

Instead of using the YUV, the standard transformation widely used is the YCbCr. Furthermore, there are a lot of libraries which implement this transformation and then call it yuv (just like the one we will use).

In the following figure we can see how localized the points classified as skin are in the projection of the two chrominance dimensions. This is the reason why we chose this color scheme as our foundation from which we built everything else.

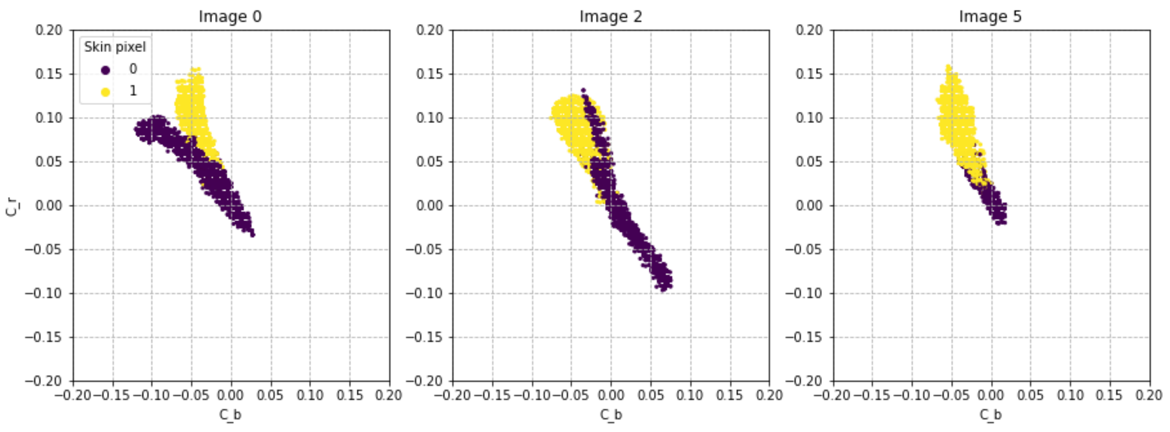

Figure 3: CbCr Projection of training images' pixel values

HSV

This color space divides the 3 dimensions of color into: Hue, Saturation and Value. This space color appeared to cover the need of a user to define the values of color, as it is more intuitive than the previous explained ones. It is the one used currently when one has to choose a color interactively instead of picking numbers.

As with the previous transformations, there's a difference between luminance (represented by the Value property) and the chrominance (represented by the Hue and Saturation components). The distinction is that it doesn't perform linear function to apply the transformation and it is why it will not serve as well as YUV representation.

Phase 2: Skin pixels detection

General scheme:

As said before, in order to predict which pixels are skin and which aren't we decided to use the characteristic Cr and Cb values that the human skin has. This presented the problem that only a pixel model was being used, therefore some inconsistencies appeared in the results. In order to remove them we used a simple post-processing based on morphological transformations.

Implementation:

As can be seen in Figure 3 the skin pixels tend to be around the same area in the 2D CbCr plane and have a similar shape, therefore we decided that, instead of using the proposed solution (a histogram), we would model the skin pixels as a gaussian distribution. This presented the disadvantage that we were limiting ourselves to a concrete distribution, therefore if we had a slightly different shape in another image (or dataset) it would be harder to perform the detection. On the other hand, this presented an advantage, since we were limiting ourselves to a gaussian, we only had 6 parameters to estimate (2 for the mean and 4 for the covariance matrix), therefore the amount of overfitting we would be performing on the training dataset would be quite small. We decided to go with CbCr instead of the HSV counterpart because after a quick test, we got a little better results with CbCr. However, one can easily change the input and it should work as well..

In order to estimate these parameters, we used all pixels classified as skin by the target mask from all training images (without taking into account from which image came each pixel). The estimated distribution can be seen, as well as some pixels from some images in Figure 4. It can be seen that the general shape of the skin region is well represented, but it seems that the mean should be slightly more to the right, or at least that's what we thought when we first saw it. After taking a closer look (and looking into more images), one could see that the density of points is higher around the mean, and that's why is predicted to be there, while on the right we have a lower density of points. This is one of the disadvantages of limiting to a gaussian distribution: we cannot catch asymmetric fluctuations from the mean.

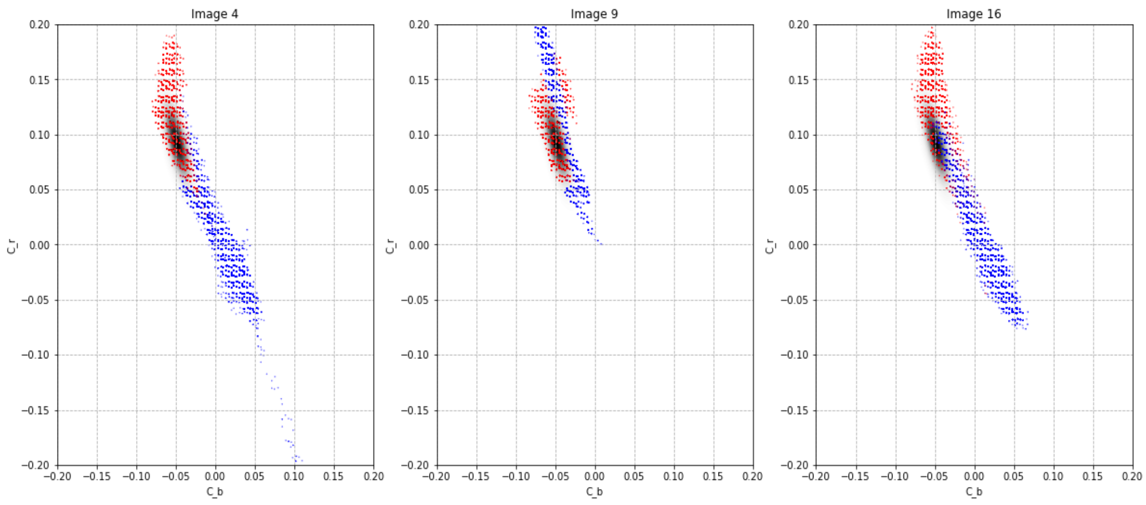

Figure 4: Gaussian Distribution (black shadow) and skin image pixels represented on CbCr

Once the distribution has been estimated from the skin pixels, in order to predict whether a pixel is skin or not, we wanted to perform a hypothesis test and decide depending on the p-values of the estimated gaussian. In order to do that we needed to integrate radially the resulting gaussian and we did not find an implementation in python. Therefore, we used the values of the pdf itself: given a point to estimate, we would calculate the pdf at that point, if the values were higher than some reference, we would predict it as skin and background otherwise. This reference was a hyperparameter: we had a threshold value between 0 and 1, such that the reference was this threshold value times the value at the mean (the pdf maximum value). Although very simple, this presented the disadvantage that, the change in reference was not linear with respect to the threshold due to the characteristic bell shape of the gaussian function, that is, if we increase the threshold from, for example, 0.1 to 0.2 we could have a much higher difference in results than if we increased it from 0.4 to 0.5.

In order to post-process the inconsistencies that appeared with the proposed scheme, we performed some morphological transformation. To remove the black pixels inside the skin, we decided to use a dual reconstruction (closing), using as a marker the edges of the image. This allowed to reconstruct the black background and to not reconstruct black areas unconnected to the background. We could have used a normal closing (dilation-erosion) , but it could have resulted in some fingers being joined and, since the final goal was to count fingers, we did not want that to happen. Apart from this, we also applied an opening with a small kernel size to remove small white regions that could have remained in the background.

Once we had already started with phase 3, we realized that we needed smoother mask edges, and using what we have explained until now, the edges were rough. So, we decided to smooth the image with a gaussian filter before binarizing and applying the threshold. The resulting images were, in general, smoother; although the roughness wasn’t solved in all of the cases.

Results:

Some good and bad examples of our predictions with and without post-processing can be seen, respectively in figures 5 and 6. For this examples a threshold of 0.15 has been used, as it will be explained later in this section.

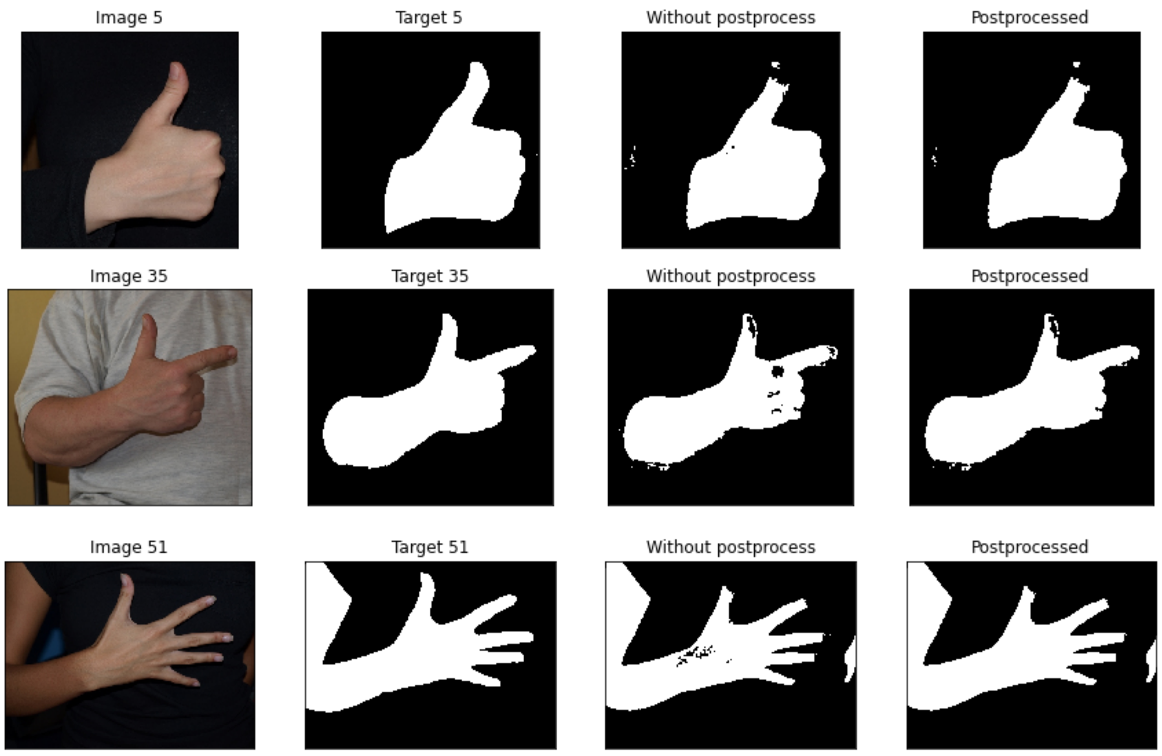

Figure 5: Good prediction examples using a 0.15 threshold.

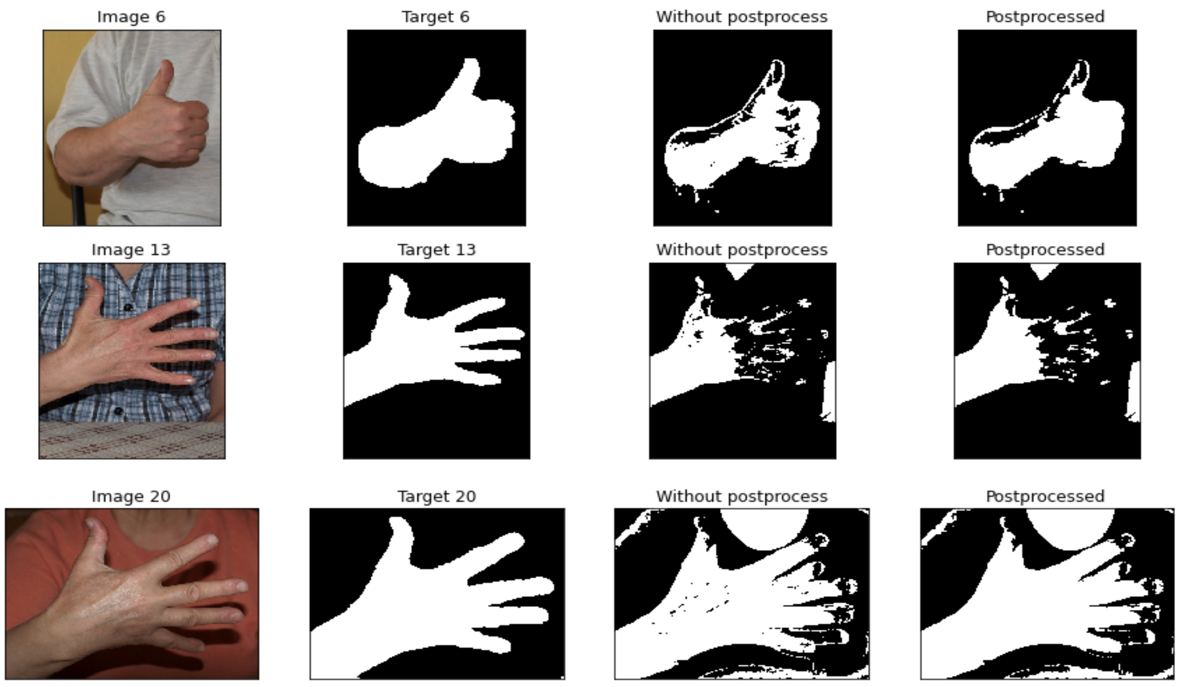

Figure 6: Bad prediction examples using a 0.15 threshold.

It can be seen that on the majority of cases, our approach yielded really good results. The combination of our gaussian skin detector with the postprocess primarily done by the hole-filling procedure achieves these satisfying results. Despite this, in some cases, the hands would not be well masked because of noise or the camera flash which made parts of the hand white, losing its chrominance values is the distinct characteristic used by our system to determine skin. Nevertheless, there were some recurrent cases whose predictions would not include the fingers no matter how reasonably low we set the threshold. As we found out by looking at the real images, all the pictures were from the same person. The difference between his/her hands and all the other hands was that his/her fingers were really pink, just like when one has their hands out in cold weather. If we take a look at the chrominance representation as well as our gaussian (Figure 4, image 16), we can clearly see that the pink values are too far apart from the gaussian center for our system to mark them as skin. This is because the majority of the skin pixels are of brownish nature mixed as well with a hint of red, but when the fingers are pink, there is no brown in it and then, our gaussian which centered around a brown point will not be able get its true skin class. If we had decided to use instead a mixture of gaussians instead of just one, we could have probably gotten those hands as well; it is a possible improvement but we decided against it for now because it could have been overcomplicating the solution. Also, it can be appreciated the curious fact that nails are not being predicted as skin, which is a really similar case than the one explained before, the nails' color tends to be pink.

Finally, there is the problem with shadows, which can be perceived as skin depending on the background color. As we can see, if the chrominance of the shadows is really similar to skin color, then our system could mistake it and classify it as skin. We could improve these results as well as all others using edge detectors (just like the canny edge detector) so to also get the spatial information of the hand. This of course would increase the complexity of the system, which is something we want to evade.

In order to formally evaluate the compare different results we decided to use two similar metrics:

- <u>F-score:</u> We would compute the F-score with all pixels, not taking into account from which image comes each pixel.

- <u>Mean F-score:</u> Since the above method may under-represent small images (we could be awfully predicting small images, but since they have less pixels the F-score won't be as affected as if the image where big). In order to solve this problem we propose to compute the F-score for each image and then average all of them.

Apart from this, we tested the general robustness of the detection algorithm by means of ROC curves and AUC, calculated varying the threshold value: a high value will yield high precision and small recall, whilst a low one would yield high recall and lower accuracy. At the end the hyperparameters were chosen by the F-score, but having a look at the AUC gave us insightful information of the general robustness of our predictions.

Since the model is very simple (it only has 6 parameter to model pixels from 60 images), we decided to perform the hyperparameter on the training test itself and see how it performs in the validation set (use it as test set). This is not the ideal, but we insist that the model is simple therefore it should not be overfitting the training set and, since the amount of data we had was limited, we thought this was a good approach.

After exploring all possible combinations of threshold and kernel size, we found that the optimal combination was a small kernel size of 3x3 and a low threshold (0.02, taking into account primarily the mean F-score), which yielded an F-score of 0.912 and mean F-score of 0.918. The fact that they are similar shows that we are obtaining similar results in small and big images, there is not a high bias present towards the distributions of big images . The ROC curve for this particular kernel size can be also seen in Figure 7. Apart from this, we saw that the kernel size did not have a huge influence on the results, whilst the threshold was the key when performing the hyperparameter optimization. This value had a direct influence on the accuracy and recall of the system, and in some cases, the general quality of predictions was determined directly by it. Depending on the application a different threshold can be used and this gives the algorithm an extra flexibility that, if used properly, makes it possible to use the same detection scheme for different application.

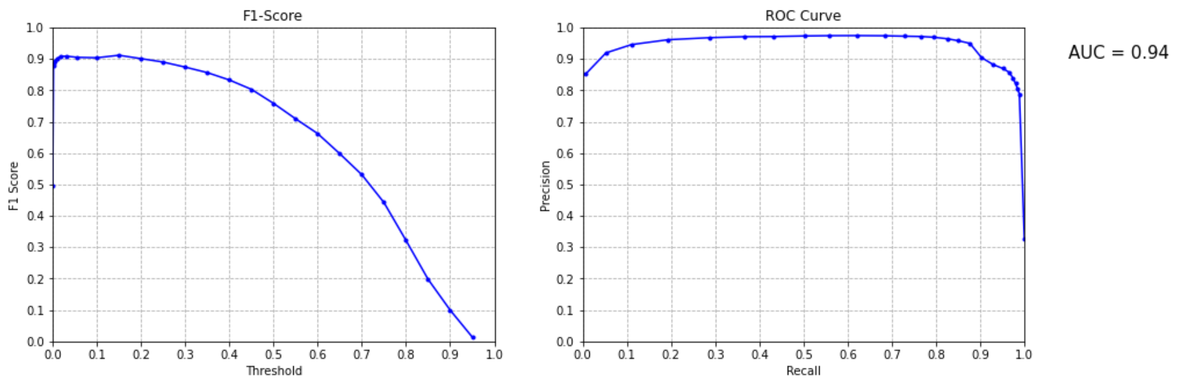

Figure 7: Training results using a kernel of 3x3.

Using this combination of hyperparameters, we tried applying this onto the validation set (which in this case we used only as testing set) and we obtained an F-score of 0.913 and a mean F-score of 0.921. The fact that they are higher values than the ones found in the training set shows that indeed we were not overfitting the training set and the hyperparameter approach was a good one. The ROC curve for the validation/test set can be seen in Figure 8.

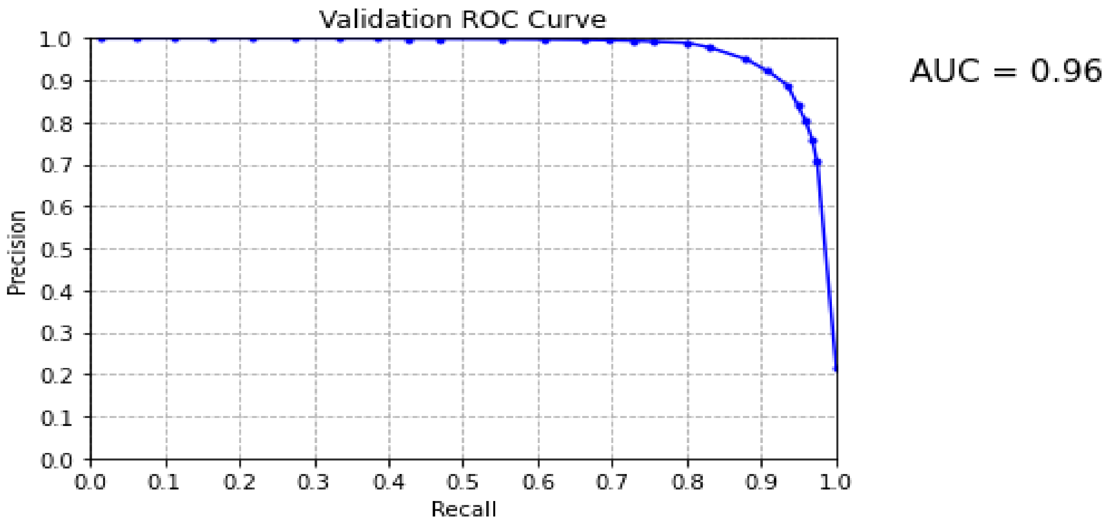

Figure 8: ROC curve for the Validation set..

Despite all this, we decided that instead of using the optimal threshold with respect to the mean F-score, we would use the other optimal threshold, the one with respect to the F-score (0.15), because it was more strict and, even so, it yielded results which had more or less the same F-score as the other one. This was more beneficial to us since we needed the outline of the fingers to be well determined instead of having the mask of the fingers join and mesh together letting our algorithm in phase 3 to have no way of differentiating between fingers.

Phase 3: Finger detection

General scheme:

For this phase, we came up with an algorithm that works in 7 steps and finds the amount of fingers in an unsupervised way and works using the masks predicted in section 2. The main idea of this is to isolate the fingers from the hand and then count the amount of isolated region. In order to do this we used the following steps:

1. Calculate minimum distance to the hand edges: In order to find where the center of the hand is, we decided to calculate, for each skin pixel, the minimum distance to the hand edge. The pixel with highest distance would be the center of the palm.

2. Hand reconstruction: Now that we had the hand center, we decided to use it as a marker to perform an opening by reconstruction and isolate better the hand region, therefore slightly improving the results of phase 2.

3. Connecting lines: By means of the edge distance, we saw that there were some lines of local maxima connecting the hand center with the fingertips (as can be seen in the first steps of the different examples in all the following figures). We decided to use this in our advantage and isolated them (and separating them from the palm center). After isolating them, we widened them in order to join some inconsistent pixels and having a better representation of these lines. How this was done will be explained in the following section.

4. Region counting: Now that we had unconnected regions corresponding to the fingers and the arm (and many ‘noise’ regions’) we only had to retrieve them.

5. Region filtering: From the obtained regions in step 4, we filtered those having a small size (‘noise regions’).

6. Check center robustness: Before applying step 7, we check whether the obtained hand center is robust, that is, we see if is close to the center of the regions obtained in step 5. If it is not robust enough, we change it.

7. Retrieve connected regions close to the hand: Finally, using the general shape of the hand and the center found in steps 1 and 6, we define an ellipse and those regions cutting with the ellipse and extending farther from its limits are taken as fingers/arm regions. Finally, we only count this regions and subtract the one corresponding to the arm.

Implementation:

The implementation of the algorithm was quite complicated especially in some steps. We will explain how we did it, but for more details we will refer you to the notebook itself (Finger Detection section). The implementation for each step goes as follows:

1. In order to find the distance to the edges, first we retrieved these edges by using a top hat (mask minus eroded mask). In order to find the minimum distance we had to 'propagate' the edges to the skin regions and count how much time it took this propagation to reach each pixel. This was done by means of a BFS (Breadth First Search) on the graph that represents the mask (we did not include the non-skin pixels in the graph) and starting it from the pixel edge. We implemented this by multiplying the adjacency matrix to the 'indicator vector' of reached nodes (that is, a flattened version of the image with 1s in the already reached pixels and 0 in the other ones). If we multiply the adjacency matrix and this indicator vector, we obtain the neighbor pixels of those indicated in the vector. We assigned as distance to the edge the amount of iterations needed to reach the pixel for the first time in the BFS (we will refer to this result as the 'distance image' and an example can be seen in all the following figures). Finally, we obtained the hand center by averaging the positions of the pixels with highest value, as well as the covariance matrix, which will be used in step 7. It is true that this implementation of the bfs is not the most efficient due to matrix multiplication, but since we wanted it to implement it in python, it was not trivial to know if the O(pixels) usual implementation would be slower (due to python fors slowness) than numpy's matrix multiplication. If we were using C++ or a different programming environment, the implementation would have been the efficient one. Despite this, we precomputed all bfs beforehand, therefore it did not suppose a problem in the hyperparameter optimization.

2. In order to obtain regions from marker pixels, in general we used opening by reconstruction (a normal one) using the marker pixels. In this case, there was only one marker pixel which was the center. Therefore we applied this to obtain the region defined by the hand.

3. Using the distance image, we wanted to only obtain the local maxima lines. We thought that it would be great to be able to threshold the distance image by an adaptive threshold depending on the local area, corresponding to the value of the local maxima in the pixel neighborhood. This would allow us to only obtain these lines and cut the connection between them and the hand center (at the 'transition region' the local maxima would be that of the hand palm, which is higher than the one of the lines running through the finger, therefore when thresholding, we set to 0 this 'transition region'). To obtain this local maxima we applied a simple dilation with a relatively small kernel size (the value of it was fixed to three and we did not optimize it as an hyperparameter, since slightly bigger sizes led to unsatisfactory results) and then we used it to threshold the distance image. It is sort of similar to the non-maxima suppression of the canny edge detectors. Finally, we widened these results by

means of another dilation with a circular kernel size (big enough to properly join the finger, but not too big, since it would join the finger to the hand). This radius defined the first hyperparameter.

4. In order to locate these unconnected regions, we used openings by reconstruction for each pixel not yet assigned to get its region and its size. The size will further on be used to filter regions.

5. Since in step 3, we dilated the resulting lines with a circle as structuring element, the sizes of the regions are proportional to it. Therefore, we decided which regions to keep based on their relative size with respect to the structuring element's area. This relative size defines another of the three hyperparameters. We later refer to this as region filtering, which we realized is a bit unfortunate since it can be a bit misleading.

6. In order to check the robustness of the center, we first calculate the mean/centroid of the pixels found in step 5. If this centroid is far from our suspected hand center we change our centroid to the closest pixel of the closest region to the centroid. Also, if the center is changed, the covariance matrix used for step 7 is changed by the covariance matrix of this regions' pixels. This is an heuristic, since we are assuming that the closest region to the centroid will be the arm and the covariance matrix will represent its shape. The new center candidate is represented as green dot in Figure 9, Step 5.

7. From the center defined in step 1 (or step 6 if we decided to move it), we draw an ellipse to check which regions intersect with it. The shape of this ellipse is found by means of the covariance matrix obtained in step 1 (or step 6 if the center has been moved). The first eigenvector of this matrix defines the direction of the major semi-axis of this ellipse, the other semi-axis is the orthogonal direction. The sizes are based on the amount of bfs levels found in step 1 (they give an idea of the hand's size) and the size of the major semi-axis is increased using a factor based on the relative sizes of the eigenvalues of the covariance matrix (the exact implementation can be seen in the code, but the general idea was to use this eigenvalue ratio but without making the ellipse too eccentric). The factor by which we multiply the above semi-axes defines the last hyperparameter. Finally, we count how many regions intersect with the edge of the ellipse to get the corresponding fingers and arm region (we are assuming that the arm, is always present).

Results:

First, we tuned the hyperparameter simultaneously on the training set and with the combination that yielded highest F-score, we tried it on the validation (that, again, we used directly as a test set). The accuracy obtained was 0.717 and the F-score 0.714. The fact that they are similar shows that we do not tend to predict best one of the classes, but predict all of them in more or less the same accuracy.

Although the final predictions were forced to be between 1 and 5, to perform the hyperparameter optimization we allowed any possible prediction, in order to not bias the

system towards big ellipse sizes for the 1 finger examples (a big ellipse removes everything and the prediction is 0, but when set to 1, it results in a correct example that should not be).
We can see in Figure 9 good training examples where this pipeline works. In contrast, in Figure 10 we have added some bad examples. All examples can be seen in the notebook.

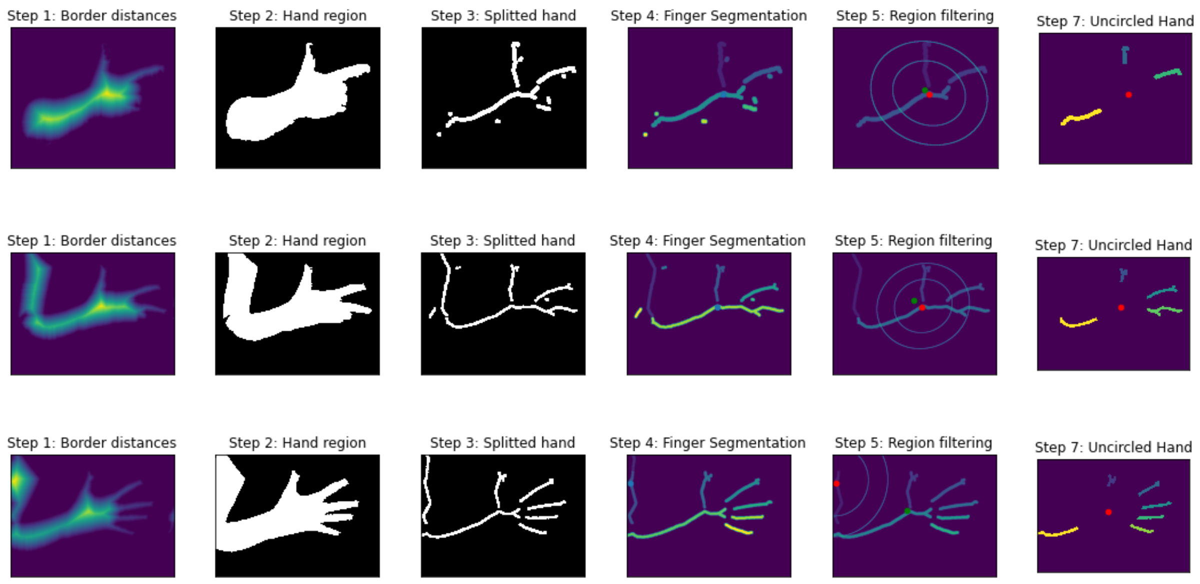

Figure 9: Good predictions. In the third one, a successful center reallocation has been performed

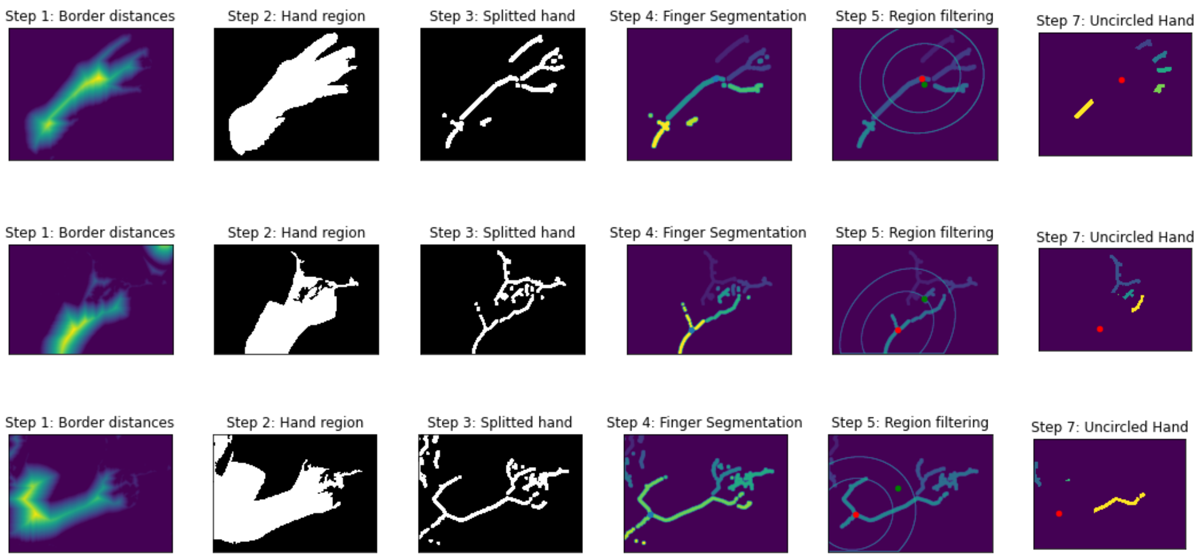

Figure 10: Bad examples. Note especially the first one where a good prediction has been performed by chance

We see that the clear difference between the good results and the bad results are the initial masks. On the former results, we start with an already nearly perfect and smooth mask, while on the latter ones, the mask is either terrible or deficient in some critical parts. For the good cases, the results are as expected, from the distances we get segmentation lines, we erase the small ones considered as noise, we change centers if necessary (if the green point which represents the mean-center is outside the large ellipse), and finally we erase the inner ellipse region and get its connected regions. As we can see, it will get the fingers and the wrist. We can also note how good the representation of the segmentation lines is, since it would also help us in getting the exact positions of the fingers. On the other hand, if the mask isn't good, we can check how the resulting segmentation lines are no good. As mentioned in the caption, the first example gets a good result nearly by chance, since its original mask has most of the middle fingers joined together. From this, we get that this system is very sensible to changes or applied noise to the mask.

When comparing the obtained results with the test accuracies, it stroke us that the values were that low, since the finger detection seemed to be doing okay (take a look at the notebook for more details). We therefore decided to take a look at the validation results and we saw that there was a major bias in the detection algorithm: we were assuming that the arm (or wrist) is always present, because so it was in the training set. This is why, although properly detecting the fingers in some validation images, the -1 made it erroneous. Since we had decided not to perform a prediction into the validation set until the very end, we did not notice this major bias. The pipeline should have a step to detect whether there is an arm or not. This will result in a major improvement into the validation accuracies. This could be done by using the bfs results and detecting whether there is a huge amount of pixels with a relatively high distance value (compared with the others). Of course this has not been implemented and we assume that it is our mistake, it is a train overfitting we have performed and does not allow the detection to generalize as well. If the training set had been different, maybe this would not have happened, but, as with machine learning, one has to be careful with this things. We have included Figure 11 in order to show this. Also, we have computed the accuracy and F1-score but removing the arm only when necessary (we marked by hand the ones that did not show the wrist), so we could show that the problem is not the algorithm itself, but the bias mentioned before and that if a proper arm detection step had been considered the results would be astonishing. The obtained accuracy was 0.804 and F1 of 0.802.

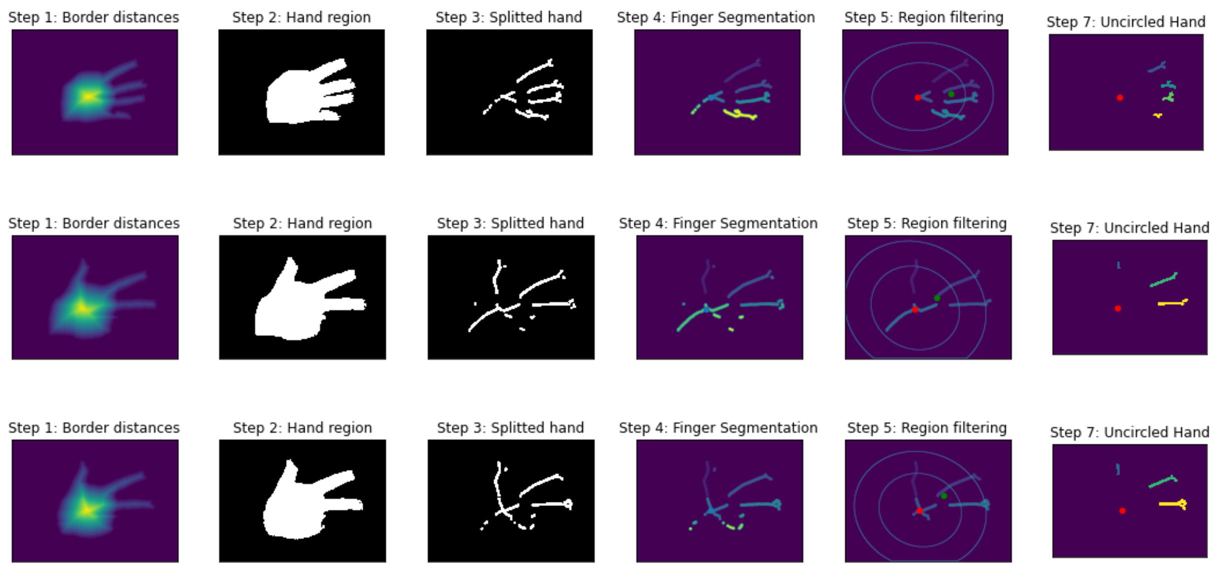

Figure 11: Validation predictions with no wrist

On another note, the sensibility to hyperparameters depended on which one was being tested. The 3rd one, the factor that multiplied the ellipse “radius”, was the most sensible out of them all, with only a difference of 0.01 one could jump a full 0.05 of fscore (which is a lot). This happened because with the initial value of 1.2, we are already detecting some fingers (specially thumbs) by a small margin, therefore an increase might result in not the detecting them. On the other hand, increasing this value allows for a better separation of fingers that might be close together leading to better accuracies. This is why this hyperparameter is the most sensible of all, but also it seems to be the most important.

Regarding the other two: The dilation kernel size was a bit unstable, since its performance was a bit ‘random’, it depended on how everything is distributed and in some images could easily join regions, whilst in others might not be regenerating them properly (leaving fingers split in two). Finally, the region parameter was quite progressive, in the sense that there is a trend when changing it. This is due to the fact that we are progressively removing more noise regions and it takes some time to start removing finger regions, and since the fingers’ sizes vary in size it starts by removing noise and then it progressively starts to remove fingers.

It is true that the hyperparameters we defined are not the most intuitive ones, specially for someone not familiarized with the system. A major improvement would be to find parameters that have a similar effect but that it’s more direct to predict how they will affect the detection algorithm.

Conclusions

After implementing the whole skin detection algorithm, we can extract several conclusions:

- It is possible to detect skin by simple pixel models by using the appropriate color features. Despite this, the presented approach may have a lack in complexity and with the correct modifications could increase in performance: the use of more complex distributions (like mixtures of gaussians or kernel density estimation) or adding to the decision process more information based on segmentation. Maybe use more features or different color spaces (or even an ensemble) could also outperform the presented scheme and be able to detect, for example, the more pinkish skin pixels.

- Following the previous conclusion, it is really interesting to note that even if the model is simple, it really performed very well. This goes a little against the actual trend where using deep learning seems to always be the answer. Of course, if we had had way more data maybe it would have been better to use deep learning tools. Nevertheless, in this case, our simple system performed really well and was a great answer taking into account to the little amount of data we had. In conclusion, one always needs to choose the tools to use depending on the situation, there is no method that outperforms the rest in all cases.

- When evaluating a prediction algorithm, not only is important to take a look at the different metrics, but also to have a look at the results. If we had only taken into account F-score for phase 2, the results of phase 3 would have been worse, because the best threshold, despite having a higher F1, did not preserve fingers as well as the one we finally used.

- It is possible to detect fingers by using general properties of the hand shape in an unsupervised way. Although the system is still missing some refinements (specially the arm problem), it shows promising results and interesting ideas that could be used for similar applications. Also it lacks some robustness in cases where the hand might be presented in another way, since we have focused in detecting fingers in the concrete images we were provided.

- It is important to check for systematic biases in data and be careful with them. In this case we saw it too late, but it serves as a lesson for future works: one has to take closer look at the data for possible bias in the data set. It also shows the importance of having an independent test set where the model has not been optimized and might not sure systematic biases with the training set.