A Practical Cosmetic Try-on System using OpenCV

Sohee Yoon

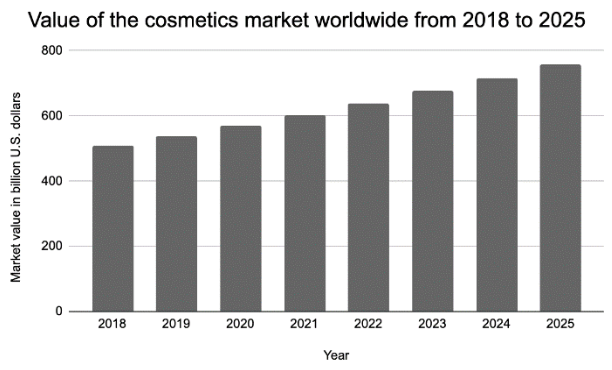
Korea International School

Abstract

There are a growing number of applications that allow customers to virtually apply cosmetic items in order to try out various makeup styles and sample new products before making an online purchase. The integration of facial recognition algorithms with augmented reality technologies facilitates this. This method has limitations due to the fact that the sampled color is shown on top of the user's skin tone without taking into account the combination of two colors and the transparency of the mixed color. Thus, the following research paper provides an alternative method for producing precise color combinations. The system comprises two primary components: First, perform gamma correction and adjust the image's brightness and contrast. The color of the user will then be extracted. Second, once the user selects the desired product color, the system mixes the user's skin tone with the color of the product. The evaluation section demonstrated the practicability of the proposed system. At the conclusion of the study, I make a heuristic proposal for a user-satisfaction-improving recommendation system based on object detection and the k-mean clustering algorithm in machine learning.

1. Introduction

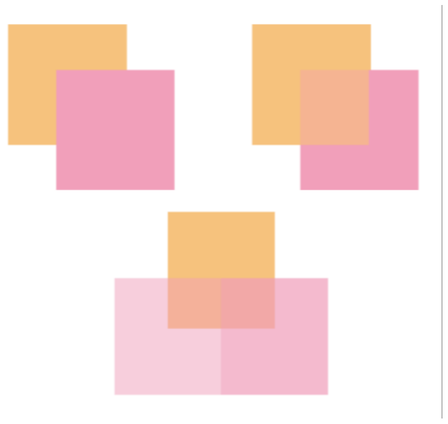
Makeup has multiple purposes, including enhancing natural features, projecting confidence and power, and expressing oneself. Makeup products grew increasingly affordable and accessible in drugstores, department stores, shopping malls, and even convenience stores as their diversity of uses expanded. As depicted in <Figure 1>, the cosmetics market's worth has climbed to $638.6 billion and is expected to continue rising. Makeup utilization is undeniably convenient, but with so many product varieties, people are bound to have difficulty distinguishing between products. The retail revolution has caused a paradigm shift in the beauty industry, with in-store makeup sampling giving way to digitized experiences.



<Figure 1>

The integration of facial recognition algorithms and augmented reality technology into beauty applications and digital services enables users to virtually apply multiple makeup and hairstyles. A ccording to a study of 3,382 women in the United States, 30% of younger customers have used a virtual beauty tool to test on makeup [1].

YouCam by Perfect Corp. has been downloaded over 900 million times worldwide; this app employs AgileFace tracking technology to let users try on hundreds of makeup looks and products. Modiface delivers an augmented reality experience for 84 beauty companies. Sephora To Go offers a virtual try-on for numerous lip product shades as well as an in-store experience [2]. In addition, numerous try-on programs display the selected color alongside the user's skin tone.



<Figure 2>

The inaccuracy of virtual try-on applications is seen in <Figure 2>. The upper left corner illustrates how most programs overlay the test color on top of users' skin, which is a different color than what users will actually see while using the test color. The illustration in the top right corner illustrates how to mix the skin color and test color with paint. The bottom depicts the combination of skin and test color with a 50% and 30% transparency filter, respectively. In this paper, I will investigate the many sorts of color expressions and demonstrate that simply combining color codes has its limits. In addition, I will propose an alternate technique that returns an exact color combination using Python color-coding, hence enhancing user satisfaction.

The preceding section introduced the paper's context, the problem to be tackled, and the limitations of the existing method. The second section describes color expressions in real and virtual environments. In Section III, the suggested system is described. The validity of the suggested system, experimental findings, and conclusions are presented in Sections IV and V. The final section of this study discusses further implications and research for this paper.

1. Prelimere

A. Color Expressions in the Real World

Paint is used to express real-life colors. Red, blue, and yellow are the primary colors; they cannot be created by mixing other colors. However, primary colors can create secondary ones by mixing. The resulting colors are known as secondary colors. Tertiary colors are generated by combining two primary colors or all three primary colors. Complementary colors are colors that, as the term implies, "complementing" one another. The complementary color of one primary color is the color created by blending the other two primary colors.

For example, the complementary color of red is green, blue orange, and yellow purple. As complementary colors, you may have noticed that the shadow of a green apple contains a bit of red [3].

B. Color Expressions in the Digital World

In the digital realm, colors are created by combining the pigments red, green, and blue. There are numerous color expression formats, including HEX, RGB, RGBA, HSL, and HSLA. Detailed descriptions of each category follow.

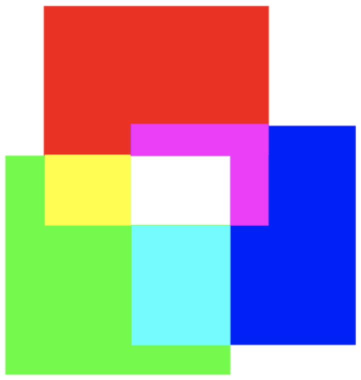
B.1. HEX

HEX represents color with a sequence of hexadecimal integers in the format #RRGGBB. The RR represents red, the GG represents green, and the BB represents blue. The range of hexadecimal integers from 00 to FF represents the level of intensity, with 00 being the weakest and FF the strongest. Recently, contemporary technology has boosted the transparency of hexadecimal integers. In standard #RRGGBB notation, there are 16,777,216 possible color combinations, as RR, GG, and BB each contain 256 distinct values between 00 and FF.

<Figure 3>'s upper left corner is colored #FF0000, a pure red, with the greatest hexadecimal values for RR (red) and the lowest for GG (green) and BB (blue) (blue). The color shown in the top right corner corresponds to #00FFFF. The bottom consists of white corresponding to the levels #FFFFFF. With the addition of transparency, the format becomes #AARRGGBB, where AA defines the level of opacity [4].

Color of #FF0000

Color of #00FFFF

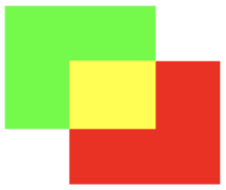


<*Figure 3*>

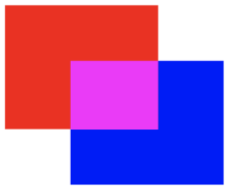
Color mixture of all three main colors: red, green, blue.

B.2. RGB

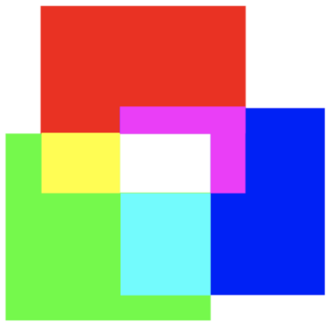
RGB refers to Red, Green, and Blue, the additive color synthesis primary colors. It has layers of red, green, and blue pixels that have been blended into a single unit. It is capable of producing 16 million colors, each with approximately 250 distinct shades [5]. The primary colors are represented by 256 levels ranging from 0 to 255. This system is utilized in display screens. The graphic above exhibits the additive color system's mixing scheme. <Figure 4>'s upper left corner displays the color yellow that corresponds to the levels R=255, G=255, and B=0. The magenta in the upper right corner corresponds to R=255, G=0, and B=255. The white at the bottom corresponds to the values R=255, G=255, and B=255.



Color of R=255, G=255, B=0



Color of R=255, G=0, B=255

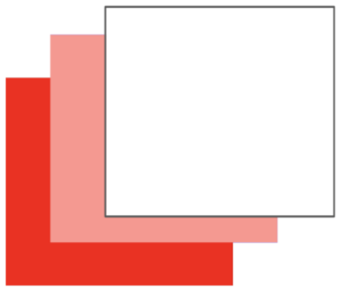


<*Figure 4*>

Color mixture of all three primary colors.

B.3. RGBA

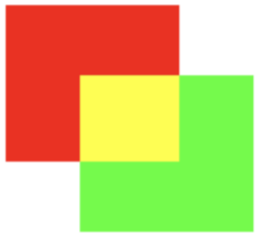
RGBA is similar to RGB (from above). The addition is made to the letter A, which represents Alpha. The first three values, Red, Green, and Blue, are represented by integers between 0 and 255. Alpha specifies the level of color's transparency. Its value is between 0 and 1: 0. 0 is fully transparent, while 1 is totally opaque.



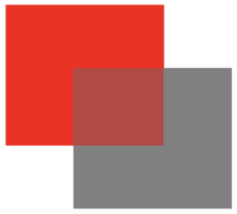
<*Figure 5*> R=255, G=0, B=255 for all three colors, and A at 1, 0.5, and 0 from left to right.

The RGB level is displayed in <Figure 5> as R=255, G=0, B=0, and A=1.0, 0.5, and 0.0, where 1.0 is the red color on the left and 0.0 is the white color on the right.

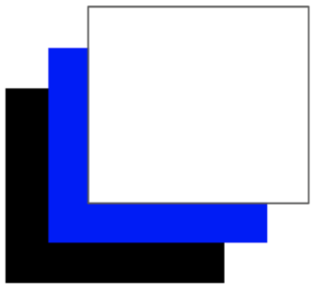
B.4. HSL



Hue is at 0° or 360°, 60°, and 120°.



Hue is at 0° or 360, and saturation is 0%, 50%, 100%.



<*Figure* 6>

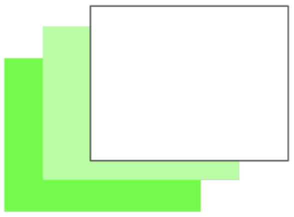
Hue at 240°, saturation at 100%, and lightness of 0%, 50%, and 100% from left to right.

HSL is an acronym for hue saturation and lightness. The hue is determined by the angle on the color wheel. Green is 120°, while blue is 240°. Saturation is quantified as a percentage, with 100% denoting complete saturation and 0% denoting a neutral gray. Lightness is also represented using percentages, with 0% signifying black and 100% white.

<Figure 6>'s upper left corner displays the mixture for hues of 0°, 360°, 60°, and 120°, which are red, yellow, and green, respectively. The top right corner displays the mixture when the hue is 0° or 360° and the saturation is 0%, 50%, and 100%. The hue is 240°, but the saturation is 100%. When the lightness is 0%, the color is black; when it is 50%, the color is blue; and when it is 100%, the color is white.

B.5. HSLA

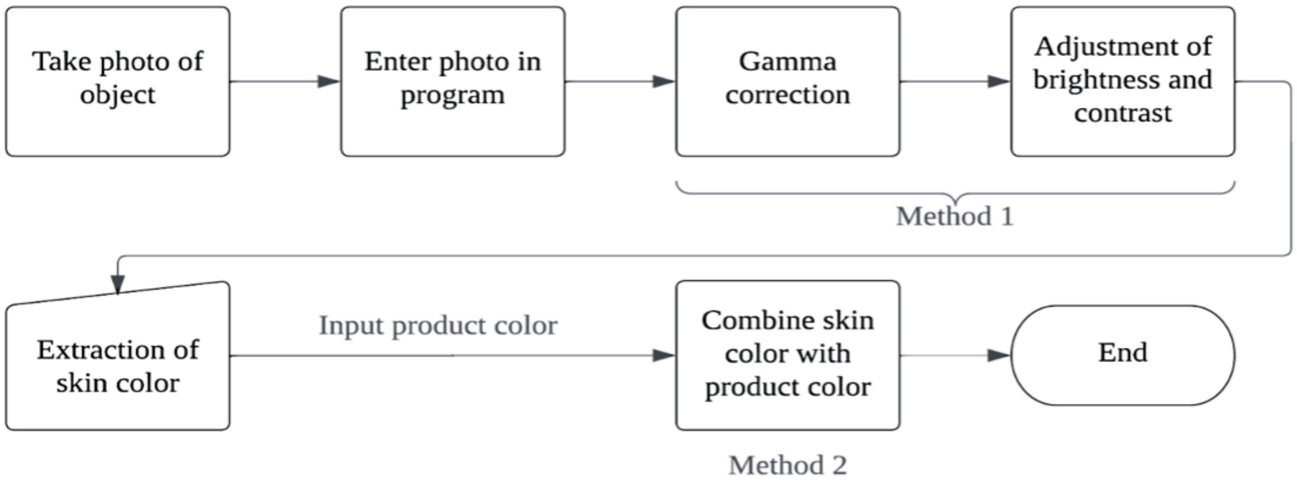
HSLA is comparable to HSL (from above). The letter A, which stands for Alpha, is what makes the difference. Alpha indicates the level of color's transparency.



<*Figure* 7> Hue at 120°, saturation at 100%, lightness at 50%, and alpha is 1, 0.5, and 0 from left to right.

The hue in <Figure 7> is 120 degrees, the saturation is 100%, and the brightness is 50%; from left to right, the alpha value changes from 1.0 to 0.5 to 0.0. The uppermost color is white, while the lowermost color is green.

II. Proposed System



<figure 8>

Virtual makeup application is mostly dependent on the synthesis of digital color-coding forms such as HEX, RGB, RGBA, HSL, and HSLA (explained above). However, the issue with existing try-on technology is that it merely blends the user's skin tone with the selected test shade through the synthesis of light, which is a far cry from reality. If all colors are merged through the synthesis of light, white is the result. In actuality, if all the hues are mixed with paint, the result is black. This decreases accuracy and user satisfaction. Therefore, I suggest a unique approach that utilizes color codes to resolve the issue of erroneous color representation in virtual beauty apps.

<Figure 8> shows a flowchart of the system I propose. The user is initially photographed on a white background. The photo is uploaded to the application, and the brightness and contrast are adjusted to accurately represent the user's skin tone.

Using gamma correction, the first method minimizes the disparity between the digital camera and human color perception. The link between photons and the digital camera is linear, as when twice as many photons are sent, the camera detects twice as strong a signal. Nonetheless, the relationship between the number of photons and the amount perceived by the human eye is nonlinear, as the human eye senses twice as many photons as a fraction brighter. The human eye is also more sensitive to changes in dark tones than bright ones.

These disparities are diminished via gamma correction. Gamma correction or Gamma is a nonlinear operation that translates the color, luminance, and tristimulus values of an image so that it most closely resembles how the human eye perceives it. The fundamental equation for gamma correction, often known as the power-law transform, is O = I ^ (1 / G). I represents the input image, whereas O represents the output image. G is the gamma value ranging from 0 to ≈4 [6].

The contrast and brightness are then adjusted using the grayscale histogram. By defining alpha and beta, contrast and brightness can be modified on the conversion scale. Consequently, the output g be:

$g(i,j) = \alpha * f(i,j) + \beta$,

where f is the input, α is the alpha value derived by dividing the intended output range of 0 to 255 by the minimum and maximum values on the accumulative grayscale, and β is the beta value calculated by plugging in $g(i,j)=0$ and $f(i,j)$=minimum gray into the equation above [7].

The actual skin color is then extracted. In method 2, the skin color and product color are transformed from RGB to CMYK based on the user's input. The opacity is then multiplied, and the colors are blended at a ratio of 1:1. The final output is the CMYK value of the user's skin tone and the color of the product.

III. Evaluation

A. Implementation

This is the implementation environment: macOS version 11.6.4, 2.2 GHz 6-core Intel Core i7 CPU, and 16GB 2400 MHz DDR4 memory. Python is the implementation language (v 3.10.2).

A.1 Libraries

I primarily utilized the OpenCV (Computer Vision Library), NumPy, and Matplotlib libraries. OpenCV employs computer vision and machine learning algorithms to accomplish a number of tasks, including identifying human actions in a video, finding similar images in a database, and enhancing image resolution. NumPy is an open-source project that makes numerical computing possible with Python. Matplotlib supports interactive, animated, and static visualizations.

A.2 Method 1 implementation

In the course of executing method 1, I defined a total of 3 functions: gamma (image, gamma) – for the optimal gamma value, convertScale (image, alpha, beta) – rearrange the scales based on the values of alpha and beta, and automatic brightness and contrast(image, clip hist percent=0.8) – abstracting

```
  def gamma(image, gamma):
    out = image.copy()
    out = ((out / 255) ** (1 / gamma)) *
  255
    out = out.astype(np.uint8)
return out
```

Line one defines the input to the function as (image, gamma). Line two creates a copy of the image and assigns it to out. Line three scales the image copy's pixels to the range 0 to 1 in order to apply the Gamma correction, and then scales back to the range 0 to 255. Line four shows the image as a

NumPy array because the operations are vectorized, allowing for quick calculation. Line five returns the output of the function.

```
    def convertScale(img, alpha, beta):
        new_img = img * alpha + beta
        new_img[new_img < 0] = 0
        new_img[new_img > 255] = 255
  return new_img.astype(np.uint8)
```

Line one defines the function's input as (img, alpha, beta) Line two assigns img x alpha + beta into new_img. Line three sets the values less than 0 as 0, which is the color black. Line four sets the values greater than 255 as 255, which is the color white. Line five returns the function's output as new_img in the NumPy form.

```
    def
    automatic_brightness_and_contrast(i
    mage, clip_hist_percent=0.8):
          gray = cv2.cvtColor(image,
    cv2.COLOR_BGR2GRAY)
        hist =
    cv2.calcHist([gray],[0],None,[256],[0,25
    6])
        hist_size = len(hist)
        accumulator = []
        accumulator.append(float(hist[0]))
        for index in range(1, hist_size):
    accumulator.append(accumulator[inde
    x -1] + float(hist[index]))
        maximum = accumulator[-1]
        clip_hist_percent *=
    (maximum/100.0)
        clip_hist_percent /= 2.0
        minimum_gray = 0
        while accumulator[minimum_gray] <
    clip_hist_percent:
            minimum_gray += 1
        maximum_gray = hist_size -1
        while accumulator[maximum_gray]
    >= (maximum - clip_hist_percent):
            maximum_gray -= 1
            alpha = 255 / (maximum_gray -
    minimum_gray)
        beta = -minimum_gray * alpha
            auto_result = convertScale(image,
    alpha=alpha, beta=beta)
        return (auto_result, alpha, beta)
```

Line one defines the function's input as (image, clip_hist_percent=0.8). Line two assigns the grayscale image as gray. Line three makes the grayscale histogram with the range 0 to 255.
Line four puts the histogram's length into hist_size. Lines five to nine create the cumulative histogram by adding the previous values of the histograms. Lines ten to twelve use the clip hist percent of 0.8 to calculate the number of pixels that needs to be clipped. Lines thirteen to sixteen clip 10% of the values from the left end of the histogram. Lines seventeen to twenty clip 10% of the values from the right end of the histogram. Lines twenty-one to twenty-two find the value of alpha with the contrast and brightness formula. Line twenty-three finds the value of beta by using

the alpha and the contrast and brightness formula. Lines twenty-four to twenty-five use the convertScale function defined previously with the alpha value in line 21 and the beta value in line 23. Line twenty-six returns the function output.

```
# main code
img_file = "images/test1.jpg"
save_file = "images/test1_result.jpg"

image = cv2.imread(img_file)
image_r = cv2.resize(image, (300, 300))
cv2.imshow('original image', image_r)
cv2.waitKey(0)

image_g = gamma(image_r, 1.5)
cv2.imshow('after gamma', image_g)
cv2.waitKey(0)

auto_result, alpha, beta =
automatic_brightness_and_contrast(i
mage_g)
print('alpha', alpha)
print('beta', beta)

cv2.imshow('after
automatic_brightness_and_contrast',
auto_result)
cv2.waitKey(0)
cv2.imwrite(save_file, auto_result)
```

Line one assigns test1.jpg from the images folder as img_file. Line two assigns test1_result.jpg from the images folder as save_file. Line three reads the img_file with OpenCV and assigns it to image. Line four resizes the image as 300 by 300 pixels and assigns it to image. Line five displays the image_r labelled as the original image. Line six displays the image_r window infinitely until any keypress. Line seven inputs image_r and the gamma value of 1.5 into the gamma correction equation and assigns it to image_g. Line eight displays the image_g labelled as after gamma.
Line nine displays the image_g window infinitely until any keypress. Lines ten to eleven input image_g into the automatic_brightness_and_contrast function defined previously and assign it to auto_result, alpha, beta. Lines fourteen to sixteen display the auto_result image labelled as automatic_brightness_and_contrast. Line seventeen displays the image infinitely until any keypress. Line eighteen saves the auto_result image.

```
import cv2
import numpy as np
import sys
np.set_printoptions(threshold=sys.maxsize)
img = cv2.imread('images/test1_result.jpg')
ycrcb =
cv2.cvtColor(img,cv2.COLOR_BGR2YCrCb)
# Cr:133~173, Cb:77~127
mask_hand =
cv2.inRange(ycrcb,np.array([0,143,77]),np.array([
255,173,127]))
```

```
img_result = cv2.bitwise_and(img, img, mask=mask_hand)
b, g, r = cv2.split(img_result)
b_sum = []
for i in range(len(b)):
    for j in range(len(b[i])):
        if b[i][j] != 0:
            b_sum.append(b[i][j])
b_sum = sorted(b_sum, reverse=True)
new_b = b_sum[:int(len(b_sum)*0.5)]
avg_b = sum(new_b)//len(new_b)
    g_sum = []
for i in range(len(g)):
    for j in range(len(g[i])):
        if g[i][j] != 0:
            g_sum.append(g[i][j])
g_sum = sorted(g_sum, reverse=True)
new_g = g_sum[:int(len(g_sum)*0.5)]
avg_g = sum(new_g)//len(new_g)
r_sum = []
for i in range(len(r)):
    for j in range(len(r[i])):
        if r[i][j] != 0:
            r_sum.append(r[i][j])
r_sum = sorted(r_sum, reverse=True)
new_r = r_sum[:int(len(r_sum)*0.5)]
avg_r = sum(new_r)//len(new_r)
print(avg_r, avg_g, avg_b)
cv2.imshow("Result", img_result)
cv2.waitKey(0)
```

Line one imports OpenCV. Line two imports NumPy as the notation np. Line three imports the system library (system-specific parameters and functions) Line four prints the full NumPy arrays. Line five reads the test1_result.jpg image from the images file and assigns it to img. Lines seven to eight convert the img from RGB to YCrCb and assign it as ycrcb. Lines eight to ten extract a specific region of the ycrcb image file and assign it to mask_hand. Lines eleven to twelve use the bitwise_and function to black out other colors except the ones in mask_hand and assign it to img_result. Line thirteen splits the image img_result's color into separate single-channel images b, g, r. Lines fourteen to twenty one gather all the b values and find the sum and the brightest 50% and then the average. Lines twenty-two to twenty-nine gather all the g values and find the sum and the brightest 50% and then the average. Lines thirty to thirty-seven gather all the r values and find the sum and the brightest 50% and then the average. Line thirty-eight prints the average of r, g, b. Line thirty-nine displays the img_result as Result. Line fourty displays the image window infinitely until any keypress.

A.3 Method 2 implementation

During the method 2 process, I wrote two functions: rgb to cmyk(r,g,b) for converting the color from the rgb code to the cmyk code and ink add for rgb(list of colours) for converting the color list to the cmyk code.

```
def rgb_to_cmyk(r,g,b):
    if (r == 0) and (g == 0) and (b == 0):
        # black
        return 0, 0, 0, cmyk_scale
    # rgb [0,255] -> cmy [0,1]
```

```
        c = 1 - r / float(rgb_scale)
        m = 1 - g / float(rgb_scale)
        y = 1 - b / float(rgb_scale)
        # extract out k [0,1]
        min_cmy = min(c, m, y)
        c = (c - min_cmy) / (1-min_cmy)
        m = (m - min_cmy) / (1-min_cmy)
        y = (y - min_cmy) / (1-min_cmy)
        k = min_cmy
        # rescale to the range [0,cmyk_scale]
        C, M, Y, K = c*cmyk_scale,
      m*cmyk_scale, y*cmyk_scale,
      k*cmyk_scale
        C, M, Y, K = int(C), int(M), int(Y),
      int(K)
  return C, M, Y, K
```

Line one defines the function rfgb_to_cmyk. Lines two to four state that if r is 0 ,g 0 and b 0, it returns 0, 0, 0 on the CMYK scale. Line five to seven rescales the RGB scale to the CMY scale. Lines eight to twelve extract the K value with the CMY values. Lines thirteen to fifteen rescale the CMYK form in the range of 0 to 1 to the range of 0 to 100 percent. Line sixteen returns C, M, Y, K

```
def ink_add_for_rgb(list_of_colours):
    C = 0
    M = 0
    Y = 0
    K = 0
    for (r,g,b,o) in list_of_colours:
        c,m,y,k = rgb_to_cmyk(r, g, b)
        C+=o*c
        M+=o*m
        Y+=o*y
        K+=o*k
    C, M, Y, K = int(C), int(M), int(Y), int(K)
    return [C, M, Y, K]
```

Line one defines the function ink_add_for_rgb. Lines two to five initialize the CMYK values to 0. Lines six to eleven individually multiply the opacity into the CMYK values. Line twelve defines the C, M, Y, K values as integers. Line thirteen returns C, M, Y, K.

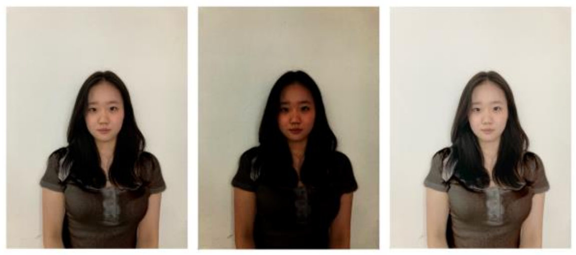
```
rgb_scale = 255
cmyk_scale = 100
r1, g1, b1 = 221, 178, 154
r2, g2, b2 = 215, 135, 112
c, m, y, k =
ink_add_for_rgb([(r1,g1,b1,0.5),(r2,g2,b2,0.5)])
print(c, m, y, k)
```

Line one assigns 255 as the rgb_scale. Line two assigns 100 as the cmyk scale. Line three is the RGB of the skin color. Line four is the RGB of the product color. Lines five to six use the function ink_add_for_rgb defined previously. R1, g1, b1 and r2, g2, b2 are inputs and the opacity value is 0.5 for both. Line seven prints the c, m, y, k value.

B. Evaluation

B.1. Gamma Correction

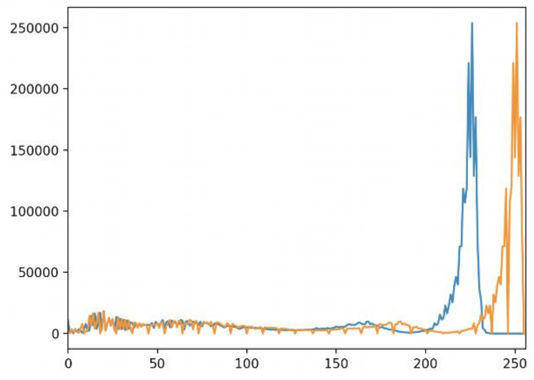
In this paper, I did gamma correction using a gamma value of 1.5. The value was determined by experiments.



<*Figure 9*> Original image, gamma value of 0.5, and 1.5, respectively.

Gamma values are in the range of 0 to ≈4. If gamma=1, then the brightness remains the same. If gamma<1, the brightness increases, and if gamma>1, the brightness decreases.

B.2. Grayscaling



<*Figure 10*> Grayscale histogram before and after clipping

The grayscale histogram is used to evaluate the different pixel intensity values of the image. The conversion formula of RGB[A] to Gray used was Y ← 0.299R + 0.587G + 0.114B. On the histogram, only 80% of the cumulatively distributed data are utilized. In <Figure 10>, the blue line represents the original cumulative histogram, while the orange line represents the clipped histogram. As illustrated in the figure above, 10% of the values are clipped from both ends so the beginning and end values of the blue line are 0.

B.3. Combining the colors

When the skin color is extracted by finding the average of red, green, and blue individually, only 50% of the brightest colors are used and the other darker colors are omitted to prevent the skin color from displaying inaccurate darkness. The skin color and product test color are combined with the opacity in a ratio of 1:1, as this ratio demonstrated the lowest percent inaccuracy.

B.4. Accuracy

I calculated the accuracy by subtracting the calculated RGB value from the real one. As the RGB distance grows, the error grows as well. The pseudocode is as follows:

Evaluation

```
rgb_scale = 256
cmyk_scale = 100

r1, g1, b1 = 221, 178, 154
r2, g2, b2 = 215, 135, 112
r3, g3, b3 = 223, 157, 133
r4, g4, b4 = 0,0,0

error = 0
t = 0.1

while t <=1:
    error = 0
    c, m, y, k = ink_add_for_rgb([(r1,g1,b1,1-t),(r2,g2,b2,t)])
    r4, g4, b4 = cmyk_to_rgb(c, m, y, k)

    error += abs(r3-r4) + abs(g3-g4) + abs(b3-b4)
    print(t, error)
    t += 0.1
```

Line one sets the RGB scale as 256. Line two sets the CMYK scale as 100. Line three is the r1, g1, and b1 values of the skin color. Line four is the r2, g2, and b2 values of the product color. Line five is the r3, g3, and b3 values of the actual product color on the skin. Line six is the r4, g4, and b4 values of the combined skin color and product color with coding. Lines seven to eight start the error value as 0, and t is the opacity. Lines nine to thirteen keep the process of finding the error going until t is 1. In lines fourteen through sixteen, the error numbers are displayed and the function is continued by increasing the increment by 0.1.

Trial	Skin color	Product color	Skin color + product color	Actual color
1	🟫 216, 176, 150	🟥 187, 82, 83	🟫 204, 129, 118	🟫 204, 120, 108
2	🟫 216, 176, 150	⬜ 248, 245, 245	🟫 235, 213, 199	🟫 231, 209, 191
3	🟫 216, 176, 150	🟫 209, 177, 156	🟫 214, 180, 156	🟫 215, 174, 142
4	🟫 216, 176, 150	🟫 158, 101, 86	🟫 189, 138, 119	🟫 184, 137, 107

Table 1: Comparison of predicted experimental results and actual results of the skin-product color combination

iter	0.4	0.5	0.6	0.7
1	38.7072	20.608000000000033	6.33599999999999985	14.668799999999962
2	6.615999999999957	11.969600000000014	38.00479999999993	54.823999999999955
3	19.502399999999966	21.652799999999957	21.652799999999957	24.363199999999978
4	36.99840000000002	19.078400000000002	10.044799999999995	21.728000000000037

Table 2: Error-values when the product color opacity is 0.4, 0.5, 0.6, 0.7.

Table 1 provides a visual representation of the color difference between the predicted and actual experimental results, which is not indiscernible but fairly slight.

Table 2 shows error values with varying degrees of opacity. The numbers in the table correspond to the product color opacity values of 0.4, 0.5, 0.6, and 0.7 and the skin color opacity values of 0.6, 0.5, 0.4, and 0.3. The lower error value indicates less error; the lowest error value in the first trial is 0.6, in the second trial it is 0.4, in the third trial it is 0.4, and in the fourth trial it is 0.6. The average lowest error value throughout all four trials is 0.5; hence, the opacity level for the product color and skin color in the experiment was adjusted to 0.5 in order to obtain the most accurate results with the lowest error value.

V. Conclusion & Future Works

In this paper, I suggested a new system for recommending cosmetic colors. This approach provides consumers with realistic color combinations that depict how a product will appear on their skin. The system's effectiveness was demonstrated during the evaluation process, as experimental results were similar to actual results.

The proposed system has a few minor flaws: the full skin color is retrieved, but not the colors of individual facial features such as the eyes or the lips, and the system cannot recommend a product because it can only display how a product appears on the user's skin.

I could adopt two machine learning methods to handle the aforementioned problems: object detection and clustering. Object detection is a type of supervised learning, and the YOLOv5 algorithm can detect particular objects. The things are categorized into different classes. Such classifications include the eyes, nose, cheeks, and lips. Thus, individual feature colors can be retrieved and mixed with product colors. Clustering is an unsupervised learning technique that groups related data. Users with similar skin tones would be grouped together and recommended products would be provided.

Reference

[1] https://segmanta.com/blog/22-women-us-used-app-try-makeup/

[2] https://www.allure.com/story/virtual-makeup-try-on-replacing-testers

[3] https://www.thesprucecrafts.com/color-theory-for-painting-2578070

[4] https://www.w3schools.com/colors/colors_hexadecimal.asp

[5] https://tachyonlight.com/rgb-led-lighting-types-benefits/

[6[https://pyimagesearch.com/2015/10/05/opencv-gamma-correction/

[7] https://docs.opencv.org/3.4/d3/dc1/tutorial_basic_linear_transform.html