Fake Image Detection Using Discrete Fourier Transform

Kim Yooseung (20215047)

May 20, 2023

Abstract

A rapid improvement in image processing has caused side effects, including proliferation of fake images. Moreover, with the support of powerful artificial intelligence, the fake image rose as one of the most serious problem in recent years. One simple but powerful approach detecting fake image is using discrete Fourier transform. In this paper, we suggest two methods for fake image detection with discrete Fourier transform, an analysis in power spectrum and using inverse Fourier transform of signal that passed through the high-pass filter. Although the former method works pretty well, we suggest to use the latter, since it can detect some fake images which cannot be detected by the former. Both methods are implemented using Python3, using some functions from `numpy`, such as `np.fft.fftshift`, `np.matmul`, `np.log` and `np.exp`.

1 Introduction

The key point of the fake detection using discrete Fourier transform analysis is that fake images lack of high frequency components. Usually, real images have high frequency components such as eyes and hair, which means that they are complicated and drastically change comparing to the neighbor pixels. However, while generating fake images, some components are crushed and become blurry, therefore high frequency components are removed and only low frequency components remain.

Utilizing this feature, we will classify an image into real if there are many high frequency components. Also, if an image has significantly less high frequency component, we will classify it into fake. The criterion for "many" and "less" will be discussed later.

2 Task 0: Prepare Image

In this paper, 7 gray scale images will be detected. So use `cv2.imread` for image loading and `cv2.IMREAD_GRAYSCALE` for gray scale.

```
# read images
images = [ cv2.imread("./images/data/00{}.jpg".format(i), cv2.IMREAD_GRAYSCALE)
            for i in range(7)]
```

3 Task 1: Perform 2D Discrete Fourier Transform

2D discrete Fourier transform is as following.

$$F(u,v) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) \exp\left[-2\pi i \left(\frac{mu}{M} + \frac{nv}{N} \right) \right] \tag{1}$$

where an image has a dimension of $M \times N$, $u = 0, 1, \ldots, M-1$ and $v = 0, 1, \ldots, N-1$. It can be implemented using two loops in Python, but it requires a number of redundant computations. Therefore, we introduce matrix multiplication for the discrete Fourier transform. First, compute the matrices that represent $\exp(\frac{mu}{M})$ and $\exp(\frac{nv}{N})$ terms, with the shape of $M \times M$ and $N \times N$, respectively. Let `exp_mu` and `exp_nv` denote the former and the latter, respectively. The implementation is as following.

```python
def exp_mu(img: np.array):
    M = img.shape[0]
    result = [[0 for _ in range(M)] for _ in range(M)]
    for u in range(M):
        for m in range(M):
            result[u][m] = np.exp(-2j* np.pi * m * u / M )
    return np.array(result)

def exp_nv(img: np.array):
    N = img.shape[1]
    result = [[0 for _ in range(N)] for _ in range(N)]
    for v in range(N):
        for n in range(N):
            result[v][n] = np.exp(-2j* np.pi * n * v / N )
    return np.array(result)
```

Now, complete the discrete Fourier transform. Let I, M_{exp}, N_{exp} denote original image, `exp_mu`, `exp_nu`, respectively. Then, the image after 2D discrete Fourier transform (F) can be represented as

$$F = M_{exp} \cdot I \cdot N_{exp} \tag{2}$$

From 2, 2D discrete Fourier transform can be implemented using `np.matmul`,

```python
def fft2(img: np.array):
    M, N = img.shape
    fourier = np.matmul(exp_mu(img), np.matmul(img, exp_nv(img)))
    return fourier
```

Then, use `np.fft.fft2` to centerize the high frequency. Also, as we only focus on the magnitude, use `np.abs` to get the magnitude of transformed image. To visualize, apply `np.log`. Therefore, the pipeline for the task1 is as following

```python
def task1(img: np.array):  # for task1 : fourier transform and visualization
    fourier = fft2(img)  # 2D discrete Fourier transform
    shifted = np.fft.fftshift(fourier)  # centerize
    plt.imshow(np.log(np.abs(shifted)), cmap="gray")  # visualize
    return shifted  # return fourier transformed (+ shifted) image
```

The result for 000.jpg is shown in Figure 1.

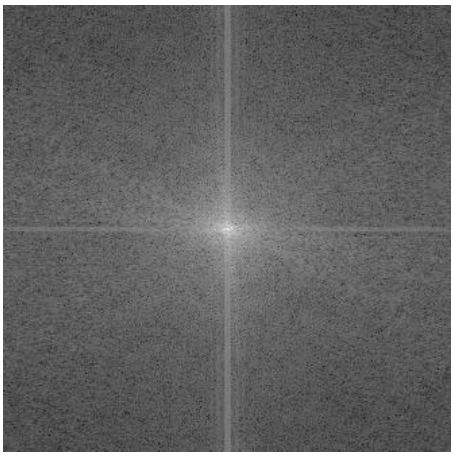


Figure 1: 000.jpg after 2D discrete Fourier transform and shifting

4 Task 2: Azimuthal Averaging

Azimuthal averaging can convert 2D power spectrum into 1D power spectrum,

$$f(\omega_k) = \frac{AI(\omega_{k+1}) - AI(\omega_k)}{\pi(\omega_{k+1}^2 - \omega_k^2)} \tag{3}$$

where $AI(\omega_k)$ is a azimuthal integration over frequency domain ϕ,

$$AI(\omega_k) = \int_0^{2\pi} \|\mathcal{F}(\omega_k \cdot \cos\phi, \omega_k \cdot \sin\phi)\|^2 d\phi \quad (k \in \mathbb{Z}^+) \tag{4}$$

As we're handling an image, who has discrete feature, the numerator of 2 becomes the sum of value of the pixel whose distance from the center is ω_k. Also, the denominator of 2 is a area difference of circles with radius w_{k+1} and w_k. Therefore, it becomes the number of pixels whose distance from the center is ω_k. The implementation for azimuthal averaging is

```
def azimuthal_averaging(img: np.ndarray):
    h, w = img.shape
    center = (h // 2, w // 2)
    # max radius (distance from the center to the corner)
    max_radius = np.hypot(center[0], center[1])
    if max_radius != int(max_radius): # if max_radius is not integer
        max_radius += 1
    max_radius = int(max_radius)  # convert to integer
    # list of sum of frequencies whose distance from the center is the same
    sum_freq = np.array([0 for _ in range(max_radius)])
    # the number of pixels with the same distance from the center
    pixels = np.array([0 for _ in range(max_radius)])
    for i in range(h):
        for j in range(w):
            # distance from the center
            radius = int(np.hypot(i - center[0], j - center[1]))
            # add pixels for given distance
            freq[radius] += img[i][j]
            pixels[radius] += 1 # add pi
    sum_freq = sum_freq / pixels  # divide into the number of pixels
    sunm_freq = sum_freq / max(sum_freq) # averaging
    return sum_freq
```

Now, plot the 1D power spectra obtained from 7 given image. The code for the task2 is as following

```
def task2():  # for task2 : azimuthal averaging
    azs = []  # 1D power spectrums
    for i in range(7):
        fft_img = np.log(np.abs(np.fft.fftshift(fft2(images[i]))))
        az = azimuthal_averaging(fft_img)
        azs.append(az)
        plt.plot(az)
    # plotting
    plt.title("1D Power Spectrum")
    plt.xlabel("Spatial Frequency")
    plt.ylabel("Power Spectrum")
    # plt.ylim(0.25, 1.2)
    # plt.xlim(-20, 250)
    plt.legend([f"00{i}.jpg" for i in range(7)])
    plt.show()
    return azs
```

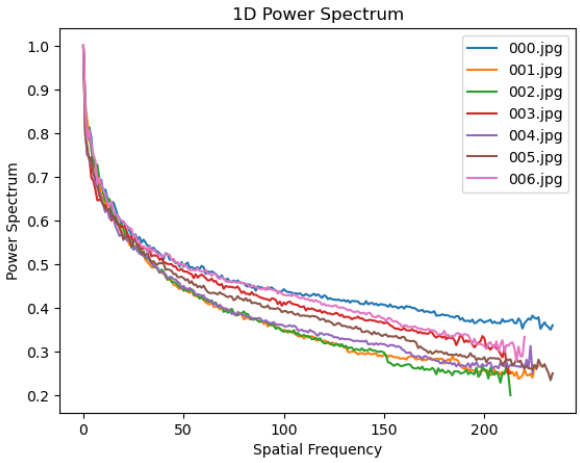


Figure 2: 1D power spectrum after azimuthal averaging

The result of `task2` is shown in Figure 2. Figure 2 shows that 001.jpg and 002.jpg have small high frequency components. Therefore, we can conclude that those two images are fake. However, 004.jpg and 005.jpg are hard to classify, since they have medium amount of high frequency components.

5 Task 3: High-pass Filtering and 2D Inverse Fourier Transform

5.1 Implementing Inverse Fourier Transform

We have detected two fake images, 001.jpg and 002.jpg. However, the criterion for the classification was not strict and there might be some deceiving images. Therefore we introduce a new method, passing through high-pass filter and apply inverse 2D Fourier transform. 2D inverse Fourier transform is computed by

$$f(m,n) = \frac{1}{M \times N} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) \exp\left[2\pi i \left(\frac{mu}{M} + \frac{nv}{N}\right)\right] \tag{5}$$

It is very similar to 1, so reuse the function `exp_mu` and `exp_nv`. Just removing minus sign on the exponent, it will yield matrix needed for the inverse Fourier transform. To specify the function if we are trying to do Fourier transform or inverse Fourier transform, add `inverse` argument to them. Thus we have the code,

```
def exp_mu(img: np.array, inverse = False):
    M = img.shape[0]
    result = [[0 for _ in range(M)] for _ in range(M)]
    for u in range(M):
        for m in range(M):
            if inverse:
                result[u][m] = np.exp(2j* np.pi * m * u / M )
            else:
                result[u][m] = np.exp(-2j* np.pi * m * u / M )
    return np.array(result)

def exp_nv(img: np.array, inverse = True):
    N = img.shape[1]
    result = [[0 for _ in range(N)] for _ in range(N)]
    for v in range(N):
        for n in range(N):
            if inverse:
```

```
                result[v][n] = np.exp(2j* np.pi * n * v / N )
            else:
                result[v][n] = np.exp(-2j* np.pi * n * v / N )
    return np.array(result)
```

Also, from 6 and similar to 2, we have

$$I = \frac{1}{M \times N} \cdot M_{exp} \cdot F \cdot N_{exp} \tag{6}$$

From 6, implement 2D inverse Fourier transform function `inverse_fft2` as following

```
def inverse_fft2(img: np.array):
    M, N = img.shape
    inv_fourier = np.matmul(exp_mu(img, inverse = True),
                            np.matmul(img , exp_nv(img, inverse = True))) / (M * N)
    return inv_fourier
```

5.2 Implementing High-pass Filter

A function `np.fft.fftshift` shifts the zero-frequency components to the center, so ones closer to the center has lower frequency. A high-pass filter removes low frequency components and only high-frequency components survive. We set the radius of the high-pass filter to 45 pixels, so pixels whose distance from the center is less than 45 will be set to zero. The implementation for the high-pass filtering is shown below.

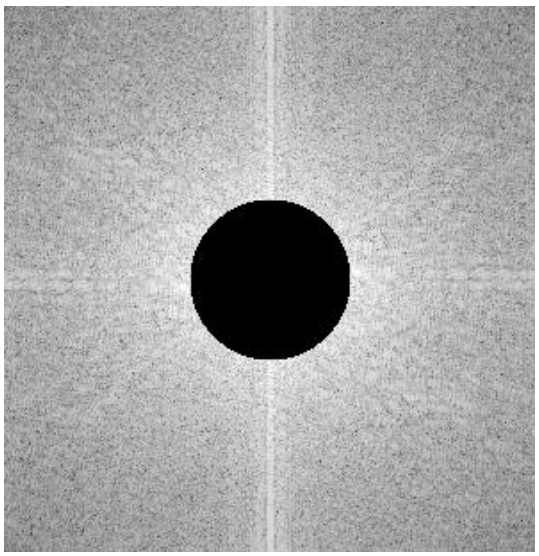
```
def highpass(img, radius = 45):
    h, w = img.shape[0], img.shape[1]
    filtered = [[0 for _ in range(w)] for _ in range(h)]
    for i in range(h):
        for j in range(w):
            # distance from the center
            distance = np.hypot((i - h // 2) ** 2 + (j - w // 2) ** 2)
            # remove low frequency components (close to the center)
            if distance < radius :
                continue
            # remain high frequency components (far from the center)
            filtered[i][j] = img[i][j]
    return np.array(filtered)
```

Combining high-pass filtering and inverse Fourier transform, we complete task3, and the code is as following.

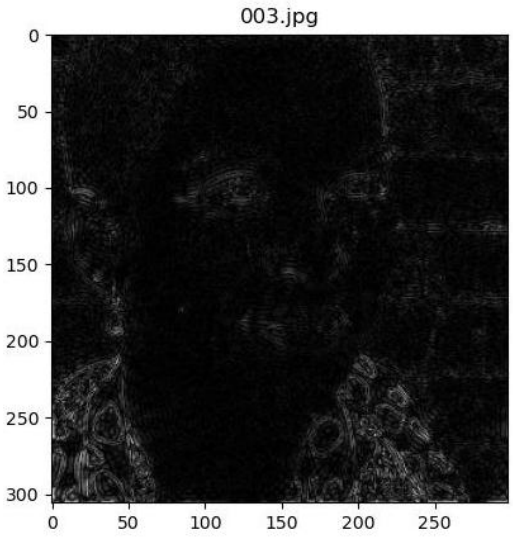
```
def task3(img: np.array):  # for task 3: high-pass filtering * inverse fourier transform
    fft_img = fft2(img)  # fourier transform
    fft_shifted = np.fft.fftshift(fft_img)  # shift
    highpass_img = highpass(fft_shifted)  # high-pass filtering
    inv_fft_img = inverse_fft2(highpass_img)  # inverse fourier transform
    img = np.abs(inv_fft_img)  # convert to real numbers
    plt.imshow(img, cmap="gray")  # visualize
    plt.show()
    return img
```

5.3 Fake Detection with Inverse Fourier Transform

Apply inverse Fourier transform implemented in Section 5.2 on to the high-passed Fourier transformed image (from `highpass()`). Then we get modified version of original image, only remaining complex parts, such as eyes, hair or some fancy decorations.



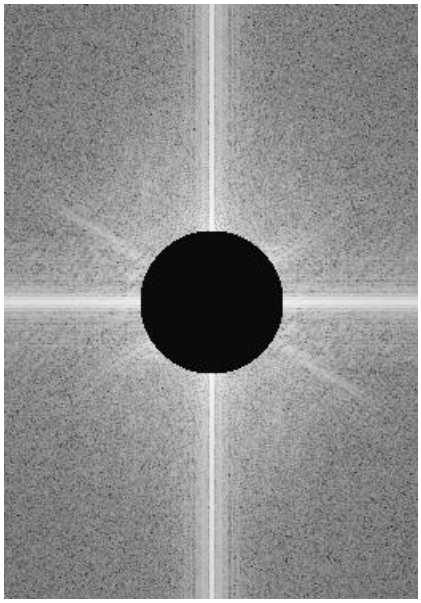
(a) Image on frequency domain



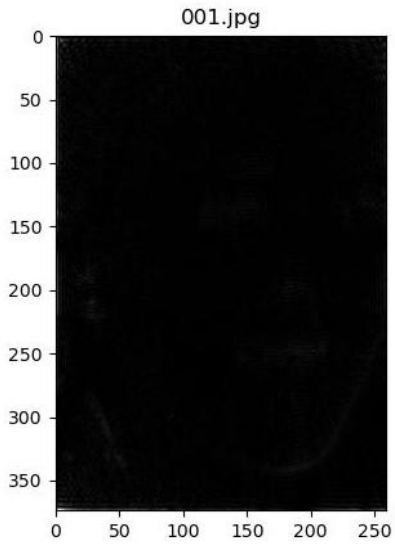
(b) Image after inverse Fourier transform

Figure 3: High-pass images of real image (003.jpg)

Figure 3(a) shows the result of high-pass filtering on a frequency domain. The black circle on the center means that low frequency components has been set to zero, or removed. Figure 2(b) shows the high-passed image after inverse Frequency transform. Most of original image was removed, but it can be observed that the eyes, nostrils, lips, hairs and earrings. Therefore, 003.jpg was classified into real image. On the other hand, almost nothing can be found in Figure 4(b). Especially, absence of the eyes, nostrils and some other features that can be found in Figure 3(b) shows that 001.jpg is a fake image.



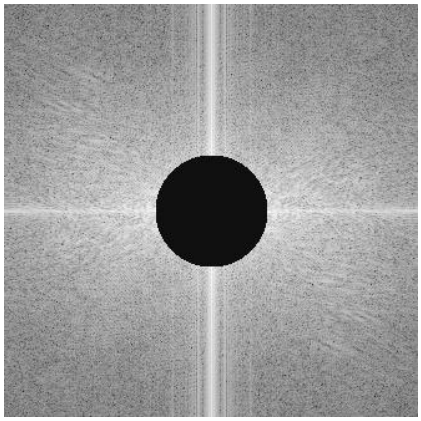
(a) Image on frequency domain



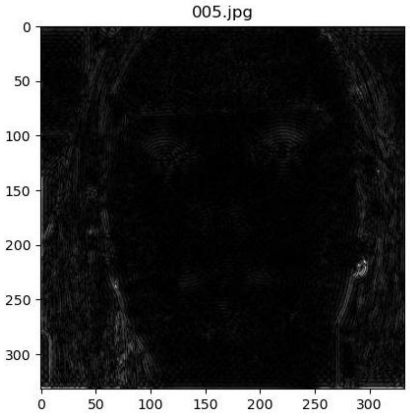
(b) Image after inverse Fourier transform

Figure 4: High-pass images of fake image (001.jpg)

Of course, our visual inspection is very intuitive and pretty accurate. However, we need some classification method or criterion for the objective result and automation. Also, the visual inspection might be deceiving, just like Figure 5.



(a) Image on frequency domain



(b) Image after inverse Fourier transform

Figure 5: High-pass images of fake image (005.jpg)

In Figure 5(b), there are some components remaining, such as hair, face contour and earrings. So we might intuitively classify into real image. It is true that those are real, but the face is fake, so no eye and nose components remaining. Therefore, we introduce a classification method for the inverse Fourier transformed image in the next subsection.

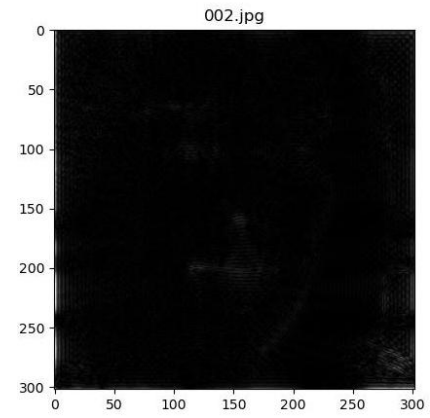
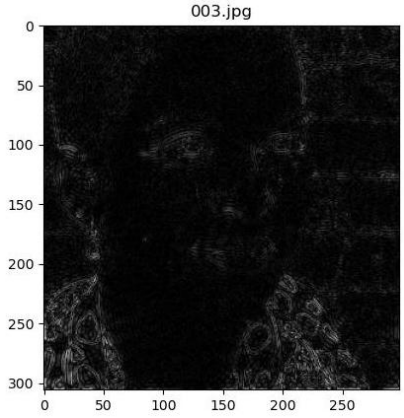
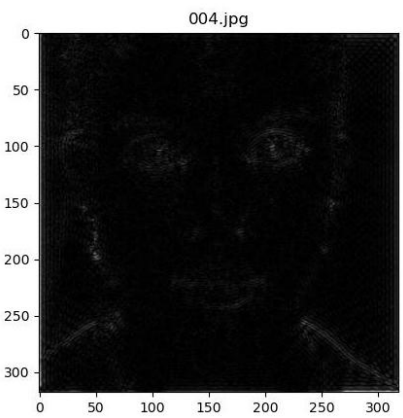
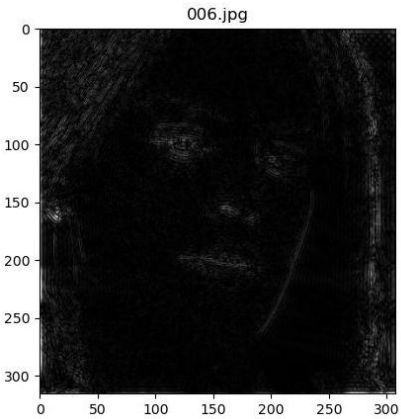
	
002.jpg	003.jpg
	
004.jpg	006.jpg

Table 1: Images after high-pass filtering and inverse Fourier transform

5.4 Classification Method

The most important component for detecting fake images is an eye. Therefore, crop the eye of the image after inverse Fourier transform. The criterion for the classification will be the mean of cropped image, since the more high-frequency components remaining, the higher the mean of the cropped image. First, list the location of eyes of given images, which was done by hand.

```
# eye location for each image
eye_location = loc = [
    np.ix_(range(70, 150), range(60, 260)),
    np.ix_(range(120, 160), range(0, 160)),
    np.ix_(range(80, 120), range(75, 225)),
    np.ix_(range(80, 130), range(75, 255)),
    np.ix_(range(80, 140), range(75, 255)),
    np.ix_(range(80, 140), range(75, 255)),
    np.ix_(range(80, 140), range(70, 230)),
]
```

Figure 6 and Table 1 shows the cropped eye image before and after inverse Fourier transform.

(a) Original Image

(b) Image after inverse Fourier transform

Figure 6: Eye cropped image (000.jpg)

After computing mean of each cropped image, now we need to cluster those values into two classes, real and fake. First, choose two centers randomly, but as we are assuming cluster with smaller center is a fake group, so assert first center is smaller than second center. Then, assign each mean to closer center and recompute the center with assigned means. Repeat until the convergence, and find the clustered centers.

```
def get_center(means, epoch=10):
    # initial centers, choose randomly, but 0 is for fake 1 is for real, so centers[0] < centers[1]
    centers = tuple(np.random.random(2).sort())
    for epoch in range(10):
        # denote which cluster each image belongs to
        assigned = [0 for _ in range(7)]
        sums = [0, 0]  # to compute new centers (average )
        for i, m in enumerate(means):
            if abs(centers[0] - m) > abs(centers[1] - m):
                # assign to the cluster with closer center, in this case, it is closer to 1
                assigned[i] = 1
                sums[1] += m
            else:
                # assigned to 0 by default
                sums[0] += m
        # compute new centers with assigned images
        new_centers = (sums[0] / (assigned.count(0) + 1e-100),
                       sums[1] / (assigned.count(1) + 1e-100))
        # add small value to avoid devision by zero error

        # update new centers until it converges
        if centers == new_centers:
```

```
            break
        centers = new_centers
    return centers
```

Now, with the clusters obtained from `get_center`, classify the images into the class that its mean is closer to. It is implemented in `detect`.

```
def detect(images, show=True):
    #save means
    means = []
    for i in range(len(images)):
        # original inverse fourier transformed image
        inv_image = task3(images[i], show=False)
        eye_cropped = inv_image[eye_location[i]]  # cropped eye image
        if show:
            plt.imshow(eye_cropped, cmap="gray")  # visualize
            plt.show()
        # compute mean, mean value of image will be used as an criterion
        means.append(eye_cropped.mean())

    # get center of two clusters, real and fake
    centers = get_center(means)
    # classify image into two clusters that is closer to its mean
    predicted = [True if abs(centers[0] - m) >
                 abs(centers[1] - m) else False for m in means]
    print(predicted)
    return predicted
```

Executing the codes above, we obtained the means of [2.77901491, 1.21486782, 1.06878478, 3.20902846, 2.42324386, 1.71739799, 2.64131109] for each images, and the center of clusters of 1.33 for fake and 2.76 for real images. Computing the distances from each means to cluster centers, we obtained the classification result as following, [False, True, True, False, False, True, False] where True is stands for real image and False stands for fake image.

6 Summary

In this paper, we introduced two methods for detecting fake images, one with 1D power spectrum and the other with high-pass filtering and 2D discrete inverse Fourier transform. 1D power spectrum was a simple way to classify the fake images, but there were some deceiving images, or some ambiguous images that are hard to discriminate. To handle this problem, we implemented high-pass filtering and 2D discrete inverse Fourier transform, so that only complex features, for example, eyes, nostrils or hair, could remain after the inverse transform. To support this method, we introduced a classification criterion. Applying both methods for 7 given images, we obtained the result, shown in Table 2.

Index	1D Power Spectrum	Filtering and 2D IFT	Conclusion
0	R	R	R
1	F	F	F
2	F	F	F
3	R	R	R
4	-	R	R
5	-	F	F
6	R	R	R

Table 2: Final fake detection table

However, there are still remaining problems in the second method. First, the given data set is so small that we cannot believe that the clustering has done well. And the cropping eyes were done by

hands, so there were errors and impossible to handle bunch of images. Using larger data set and eye detection program so it crops eyes automatically can be the solution.