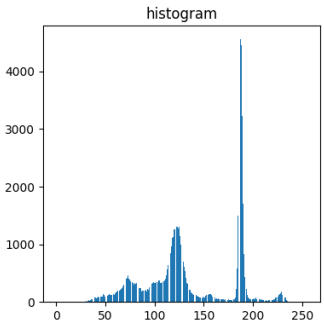
영상처리 Project 2

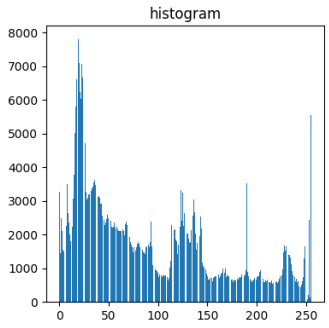
2016121150 윤준영

1. K-means / Mean shift

1.0) Original images.

본 과제에서 사용한 영상과 각 영상의 히스토그램은 다음과 같다.





<house> *<FN1>*

1.1) Implement the k-means and Mean shift algorithms.

1.1.a) K-means

K-means 알고리즘은 먼저 k개의 sample을 선택한 후, 나머지 데이터에 대해 거리(RGB값의 차이)를 계산한 후 가장 가까운 그룹으로 분류한다. 분류된 그룹에 대해 각각 평균값을 구한 후, 구한 평균값을 각 그룹의 대푯값으로 지정된다. 지정된 대푯값과의 거리를 통해 다시 데이터를 분류한 후, 분류된 데이터들의 평균값을 각각 구한다. 위 과정을 반복하며, 분류된 데이터 라벨이 이전 반복의 결과와 같을 경우 반복을 정지한다.

이미지를 처리의 거리를 계산할 때, 각각의 거리를 계산하게 되면, 반복횟수가 많아져 계산시간이 증가함을 확인할 수 있었다. 따라서 행렬로 거리를 계산하는 함수 distmat을 작성하였다. [y,x,k]의 크기를 가지는 행렬을 생성하여, 원래 이미지와 k=1, k=2, ...의 차이를 행렬로 구하고, 3번째 차원에서의 최솟값을 가지는 인수를 이용하여 분류하였다.

```
def distmat(x,y):
### 3D distance, Matrix ver. ###
    d = np.sum((x-y)**2,axis=2)
    return d
```

K-means 알고리즘을 이용하여 작성한 함수의 주요 부분은 다음과 같다.

```
def kmeans_image(rgb, k):
…
…
    # set initial grouping point and calculate distance map (y, x, k)
    for i in range(k):
        g[i,:] = rgb[rany[i], ranx[i], :]
        dis[:,:,i] = distmat(rgb, g[i])
    # initial grouping with distance, save new index > nind
    nind = np.argmin(dis, axis=2)
    # repeat until old and new index array is same
    while(not(np.array_equal(nind,ind))):
        # make new index to old, group initialize > ind
        ind = np.copy(nind)
        # make new grouping point with grouped data > g
        for i in range(k):
            if rgb[ind==i].size != 0:
                g[i,:] = np.average(rgb[ind==i],axis=0)
        # grouping with distance, save new index > nind
        for i in range(k):
            dis[:,:,i] = distmat(rgb, g[i])
        nind = np.argmin(dis, axis=2)
    sort = np.zeros_like(rgb)
    for i in range(k):
        sort[ind==i] = g[i]
    return nind, sort
```

1.1.b) Mean shift

Mean shift 알고리즘은 선택한 sample에 대해 주변의 데이터(RGB값)와 비교하여 비슷한 데이터를 가진 주변값을 이용하여, 주변값들의 mean값이 향하는 방향으로 shifting하는 방법이다. Shifting을 반복하게 되면 샘플의 local minimum point에서 수렴하게 된다. 주변의 무게중심으로 이동한다고 생각할 수 있다. Shifting를 하기 위한 mean shift값은 gaussian kernel 등의 kernel을 주로 사용한다. 마지막으로 수렴된 점을 지정하는 h값 이내에 있는 점을 모아 군집화한 후 군집들로 각각의 데이터를 분류한다.

K-means 알고리즘과 마찬가지로 kernel을 사용할 때, 속도 향상을 위하여 gaussian kernel map을 구하는 함수를작성하였다. 또한, mean shift과정에서는 속도 개선을 위해 s배로 downscaling한 이미지를 사용하였다. Gaussian kernel은 다음과 같이 표현할 수 있으며, 작성한 함수는 다음과 같다.

$$gaussian\ kernel: k(\boldsymbol{x}) = \begin{cases} e^{-||x||^2}, & ||\boldsymbol{x}|| \le 0 \\ 0, & ||\boldsymbol{x}|| > 1 \end{cases}$$

```
def gk_map(x):
### Gaussian kernel ###
### x : vector
    norm = np.linalg.norm(x,axis=1)
    gk = np.zeros_like(norm)
    gk[norm<=1] = np.exp(-norm**2)[norm<=1]
    return gk
```

```
def downscale(img, n):
    ### make 1/n image ###
    result = np.zeros((img.shape[0]//n, img.shape[1]//n, img.shape[2]))
    img = img[:result.shape[0]*n, :result.shape[1]*n]
    d = 0
    for i in range(n):
        for j in range(n):
            result += img[i::n,j::n]
            d = d+1
    result = result/d
    return result.astype(np.int16)
```

Mean shift 알고리즘을 이용하여 작성한 함수의 주요 부분은 다음과 같다. 1. 수렴점 탐색, 2. 수렴점 군집화, 3. 데이터 분류의 3가지 단계로 이루어져 있다.

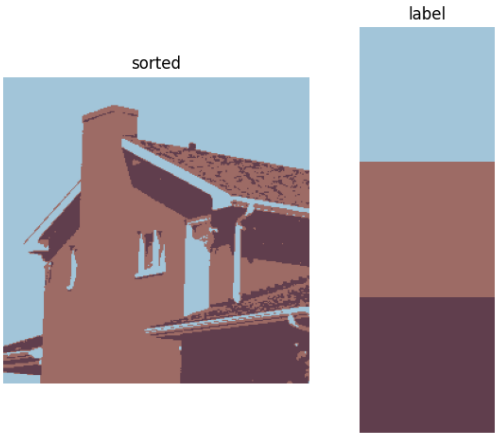
```
def meanshift_image(rgb, h=50, s=4):
...
...
    # n : no of samples, x : sample vector, y : current shifting point
    # v : convergence point, E = epsilon, c : cluster, c_center : center of cluster
    ## 1. find convergence point v with downscaled image
    rgb_ds = downscale(rgb, s)
    x = rgb_ds.reshape([n, rgb_ds.shape[2]]).astype(np.int16)
    for i in range(n):
        # calculate y[t+1] from y[t] and x[i]
        y = np.copy(x[i])
        e = E+1
        while(e>E):
            gk = gk_map((x-y)/h).reshape([n,1])
            y_new = y + np.sum(gk * (x-y), axis=0)/ np.sum(gk)
            e = dist(y,y_new)
            y = y_new
        # save convergence point to v
        v[i] = y
    ## 2. grouping v
    c_new = np.copy(c)
    i = 0
    while(1):
        # pick random unclustered sample
        rest = np.asarray(np.nonzero(c==i))[0,:].tolist()
        ran = sample(rest,1)
        c_center.append(v[ran])
        # regroup to I cluster and others(i+1)
        for j in rest:
            if dist(v[j],c_center[i]) < h : c_new[j] = i
            else : c_new[j] = i+1
        # if there's no i+1 group; break
        if (c==c_new).min() :
            i=i+1
            break
        c_center[i] = x[c_new==i].mean(axis=0)
        c = np.copy(c_new)
        i = i+1
```

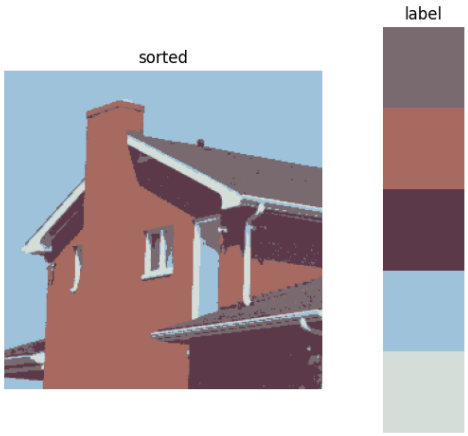
```
    ## 3. grouping image with c_center
    sort = np.zeros_like(rgb)
    dis = np.zeros(rgb[:,:,0].shape + (i,))
    for k in range(i):
        dis[:,:,k] = distmat(rgb, c_center[k])
    ind = np.argmin(dis,axis=2)
    for k in range(i):
        sort[ind==k] = c_center[k]
    c_center = np.vstack(c_center).reshape([i,1,rgb.shape[2]]).astype(np.int32)
    return ind, sort
```

1.2) Cluster the image based on the (R, G, B) vector of the image, and visualize.

1.2.a) K-means with (R, G, B)

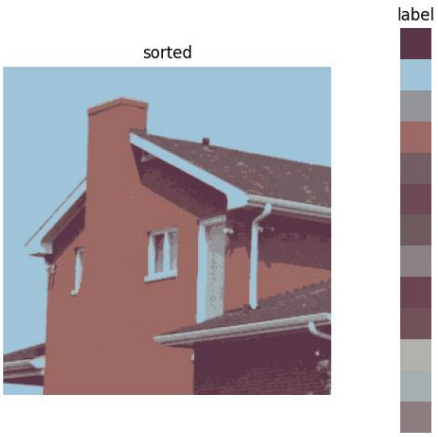
R, G, B값을 이용하여 k-means 알고리즘(k=3, k=5)을 사용한 결과는 다음과 같다.

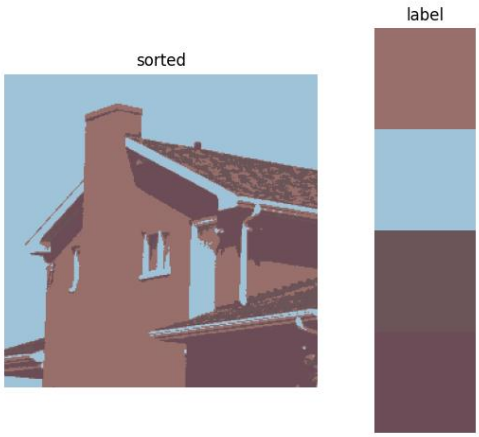




1.2.b) Mean shift with (R, G, B)

R, G, B값을 이용하여 mean shift 알고리즘(h=40, h=50)을 사용한 결과는 다음과 같다.





K-means 알고리즘을 사용할 때에는 k값에 따라 이미지에 관계없이 k개의 색으로 군집화하는 것에 비해 mean shift 알고리즘은 h값을 지정하면 이미지에 따라 극값을 찾아 자동으로 군집화하는 색의 개수를 찾아서 분류한다.

1.3) Convert RGB to YUV image, and apply k-means and mean-shift algorithms.

다음 행렬을 이용하여 RGB 이미지를 YUV 이미지로 변환한 후 군집화를 수행하고, 역행렬을 이용하여 다시 RGB 이미지로 변환하였다. RGB 이미지를 YUV 이미지로 변환하는 함수와 YUV 이미지를 RGB 이미지로 변환하는 함수를 작성한 후 k-means 함수와 mean shift 함수에 color 변수를 추가하여 YUV 이미지로 군집화를 수행할 수 있는 옵션을 추가하였다.

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}, \quad \begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.140 \\ 1 & -0.395 & -0.581 \\ 1 & 2.032 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix}$$

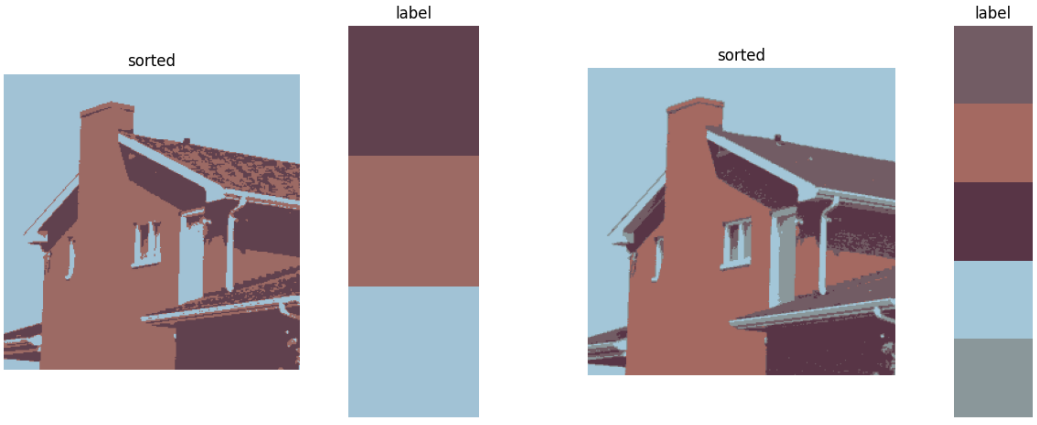
```
def R2Y(rgb):
    yuv = np.zeros_like(rgb).astype(np.int16)
    yuv[:,:,0] = 0.299*rgb[:,:,0] + 0.587*rgb[:,:,1] + 0.114*rgb[:,:,2]
    yuv[:,:,1] = - 0.147*rgb[:,:,0] - 0.289*rgb[:,:,1] + 0.436*rgb[:,:,2]
    yuv[:,:,2] = 0.615*rgb[:,:,0] - 0.515*rgb[:,:,1] - 0.100*rgb[:,:,2]
    return yuv

def Y2R(yuv):
    rgb = np.zeros_like(yuv)
    rgb[:,:,0] = yuv[:,:,0] + 1.140*yuv[:,:,2]
    rgb[:,:,1] = yuv[:,:,0] - 0.395*yuv[:,:,1] - 0.581*yuv[:,:,2]
    rgb[:,:,2] = yuv[:,:,0] + 2.032*yuv[:,:,1]
    rgb = np.clip(rgb, 0, 255)
    return rgb.astype(np.uint8)
```

다음은 k-means 함수에 color 변수를 추가한 예시이다.

```
def kmeans_image(rgb, k, color='rgb'):
    if color == 'yuv': rgb = R2Y(rgb)
...
    if color == 'yuv':
        sort = Y2R(sort)
        g = Y2R(g)
...
```

1.3.a) K-means with (Y, U, V)

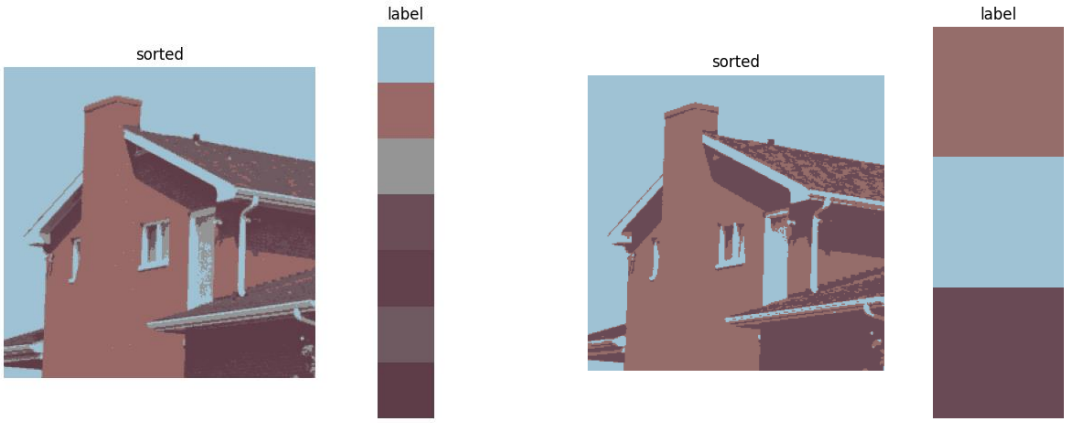
Y, U, V값을 이용하여 k-means 알고리즘(k=3, k=5)을 사용한 결과는 다음과 같다.



YUV 값을 이용하여 k-means 알고리즘을 사용한 결과, RGB 값을 이용할 때와 유사한 결과를 얻을 수 있었다.

1.3.b) Mean shift with (Y, U, V)

Y, U, V값을 이용하여 mean shift 알고리즘(h=30, h=35)을 사용한 결과는 다음과 같다.



YUV 값을 이용하여 mean shift 알고리즘으로 분류한 결과, RGB 값과 같은 h 값을 사용하면 label의 개수가 매우 적어져서 h 값을 의도적으로 낮추어 분류하였다. 이를 통해 해당 이미지의 YUV 값의 스펙트럼이 넓지 않다는 것을 알 수 있었으며, 이미지에 따라, 색상 분류에 따라 h 값을 조정해야 한다는 것을 알 수 있었다.

1.4) Use the position information (X, Y) in addition to the (R, G, B) or (Y, U, V) for segmentation. Compare performance of each case.

색상 값과 더불어 좌표를 추가하여 군집화 하게 되면 주위에 있는 색상이 조금 다르더라도 가까운 위치에 있으면 같은 그룹으로 묶을 수 있으며, 색상이 같아도 위치가 많이 떨어져 있으면 다른 그룹으로 묶일 수 있다. 이는 색상이 많이 존재할 경우 다른 개체이지만 같은 그룹으로 묶이는 경우를 방지할 수 있다.

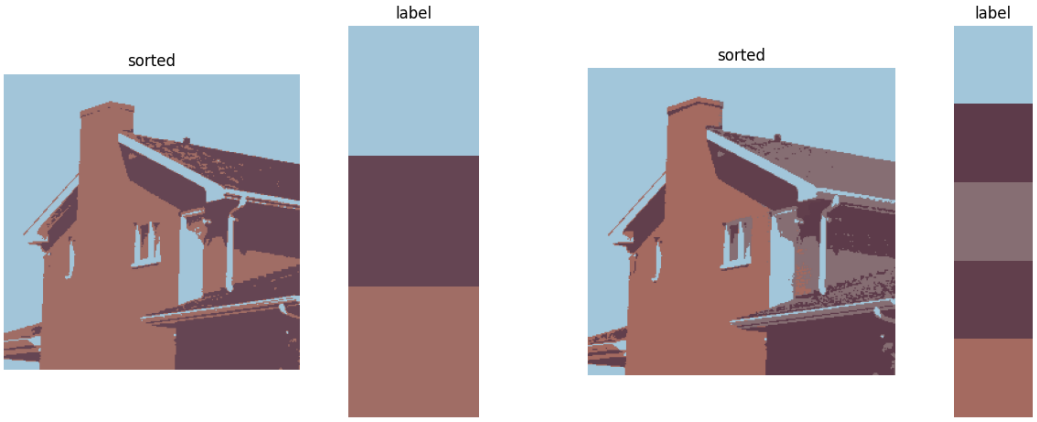
K-means 함수와, mean shift 함수는 입력 행렬의 차원과 상관없이 작동하도록 작성되어 있으나, pos 변수를 추가하여 위치 정보를 함수 내에서 추가하고, 위치 벡터의 크기를 scale하여 위치의 영향을 조절할 수 있도록 하였다. 다음은 k-means 함수에 pos 변수를 추가한 예시이다.

```
def kmeans_image(rgb, k, color='rgb', pos='x'):
    if pos != 'x' :
        yp, xp = np.meshgrid(np.arange(rgb.shape[0]), np.arange(rgb.shape[1]))
        rgb = np.dstack((rgb, np.transpose(yp)/pos, np.transpose(xp)/pos))
...
    if pos != 'x':
        sort = sort[:,:,:3].astype(np.uint8)
        g = g[:,:,:3].astype(np.uint8)
...
```

1.4.a.1) K-means with (R, G, B, Y, X)

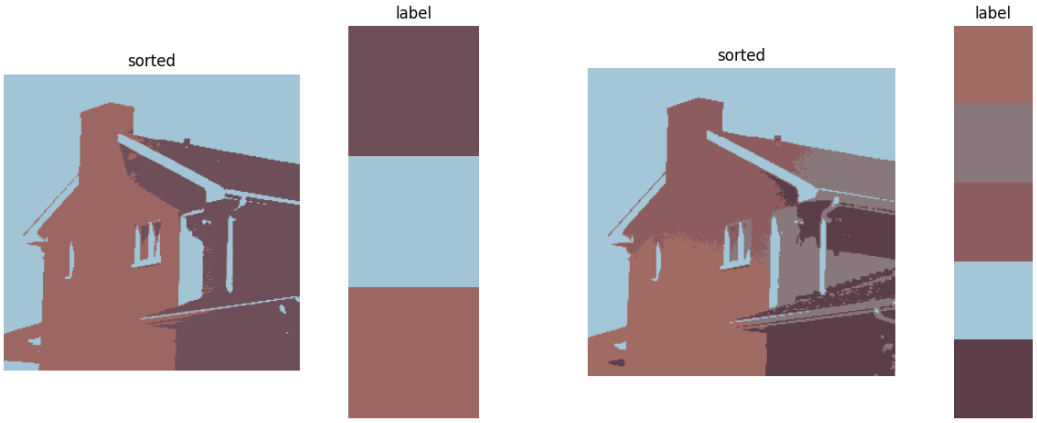
RGB 색상과 위치 정보를 이용하여 k-means 알고리즘을 적용한 결과 다음과 같다. 다음 분류된 이미지를 보면, k=3의 경우, 오른쪽 아래층 지붕 부분이 위치 정보를 이용하지 않았을 때에는 두가지의 색으로 분류되었지만, 위치 정보를 이용하면 거의 한 색으로 분류된 것을 확인할 수 있었다. k=5의 경우에는 집의 오른쪽 벽이 지붕과 같

은 색으로 분류된 것을 확인할 수 있었는데 이는 위치 정보를 이용하여 분류가 잘못된 것을 확인할 수 있었다.



1.4.a.2) K-means with (Y', U, V, Y, X)

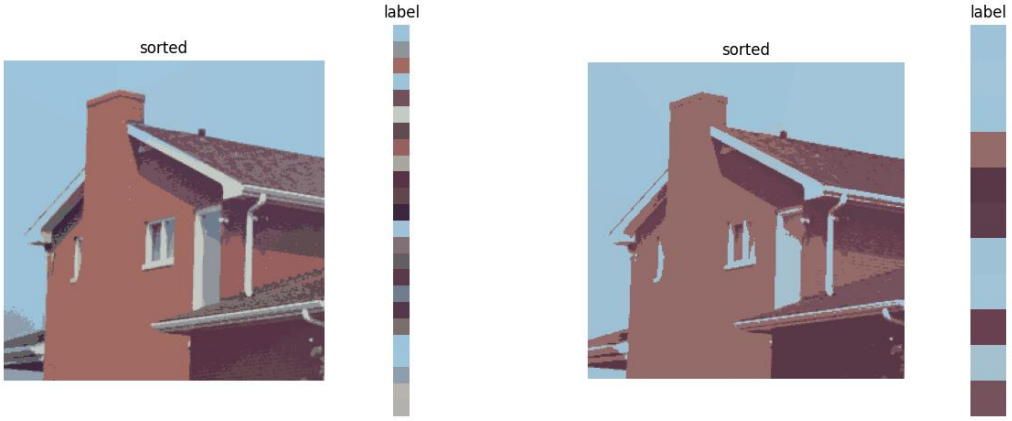
다음은 YUV 색상에 위치 정보를 추가하여 k-means 알고리즘(k=3, k=5)을 적용한 결과이다.



위 분류된 이미지를 보면 k=3의 경우, 집의 왼쪽 부분과 오른쪽 부분, 하늘, 이렇게 3가지로 분류된 것을 확인할 수 있었는데 이는 위치 정보를 사용하여 오히려 잘못 분류된 것을 알 수 있다. k=5의 경우 집은 4가지 영역으로 분류되었지만 잘못 분류되었다는 것을 육안으로도 확인할 수 있었다.

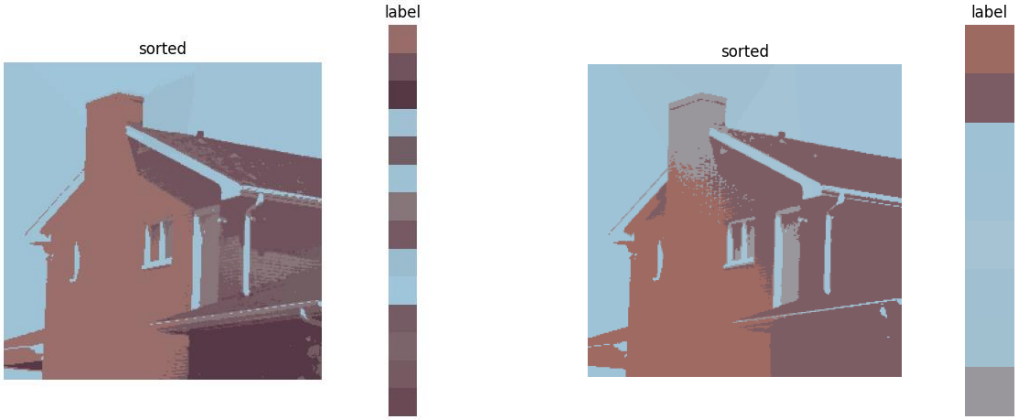
1.4.b.1) Mean shift with (R, G, B, Y, X)

RGB와 YX정보를 이용하여 mean shift 알고리즘(h=50, h=65)으로 분류한 결과는 다음과 같다.



1.4.b.2) Mean shift with (Y', U, V, Y, X)

YUV와 YX정보를 이용하여 mean shift 알고리즘(h=50, h=65)으로 분류한 결과는 다음과 같다



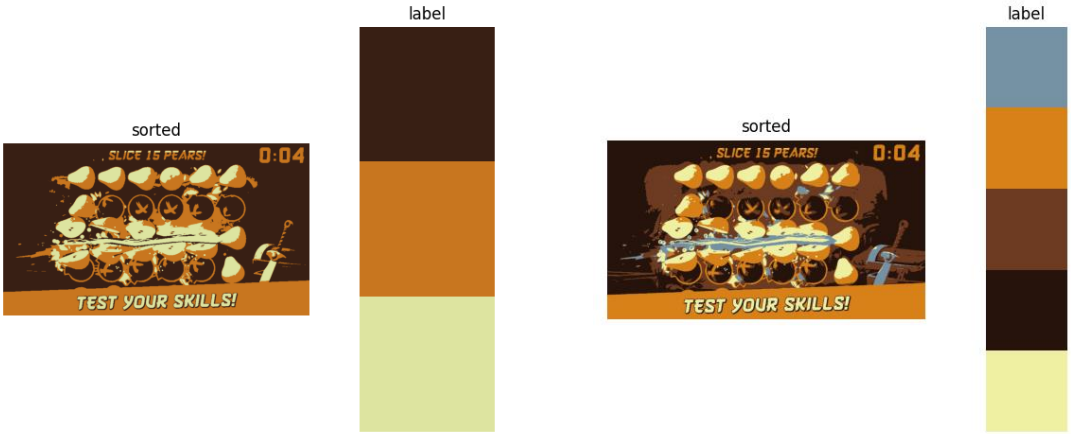
YUV와 위치 정보를 이용하여 분류한 결과, 해당 영상에서는 K-means와 mean shift를 이용할 경우 모두 분류가 잘 안되는 것을 확인할 수 있었다. 따라서 위치 정보를 이용하는 경우 계산의 복잡성은 증가하지만 항상 분류에 도움이 되지 않고 오히려 악영향을 끼칠 수 있다는 것을 확인할 수 있었다. 위치 정보를 이용하지 않을 경우를 잘 판단하여야 한다는 것을 알 수 있었다.

분류에 사용한 영상의 histogram을 보면 색상이 크게는 이분화 되어있고, 작게는 삼분화 되어있어 분류 알고리즘을 사용하여 분류할 때 k-means의 k 값을 작게 하거나, mean shift의 h 값을 간단하게 분류하여도, 매우 분류가 잘되는 것을 확인할 수 있었다.

1.5 Repeat the image segmentation for cartoon of game images.

1.5.a.1) K-means with (R, G, B)

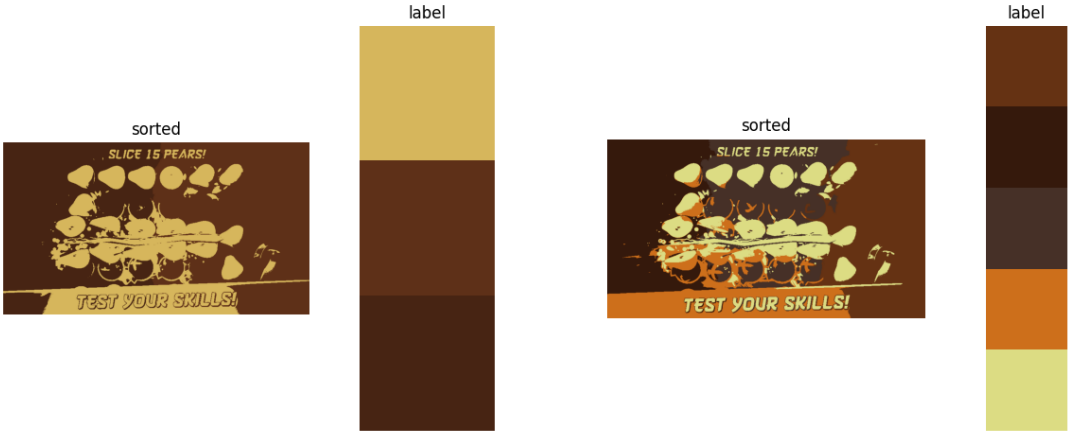
다음은 게임 이미지의 RGB 값을 이용하여 k-means 알고리즘을 통해 분류한 결과이다.



k=3으로 분류할 경우, 배경, 과일의 그림자, 과일로 분류된 것을 확인할 수 있었다. 폭탄 부분의 X 글자 또한 과일의 그림자와 같은 색으로 분류되었다. 5개의 그룹으로 분류할 경우, 과일과 칼, 두개의 배경, 과일의 그림자로 분류되었다. 마찬가지로 폭탄의 X 부분은 과일의 그림자와 같은 그룹으로 분류되었다. 과일의 그림자와 폭탄의 X 글자의 색이 유사하기 때문에 같은 그룹으로 분류된 것을 확인할 수 있다.

1.5.a.2) K-means with (R, G, B, Y, X)

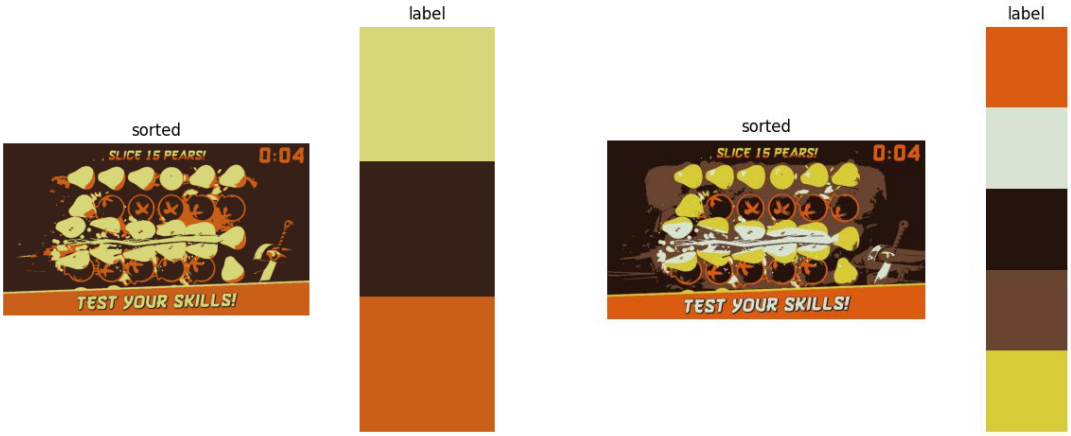
다음은 게임 이미지의 RGB 값과 YX 값을 이용하여 k-means 알고리즘을 통해 분류한 결과이다.



RGB와 위치 정보를 이용한 결과, k=3일 때에는 과일과 두 배경색으로 분류된 것을 확인할 수 있었다. 폭탄은 배경과 같은 색으로 분류되었다. k=5일 때에는 일부 폭탄의 테두리는 배경과 같이 분류되었고, 일부는 과일의 그림자와 같은 색으로 분류되었다.

1.5.a.3) K-means with (Y, U, V)

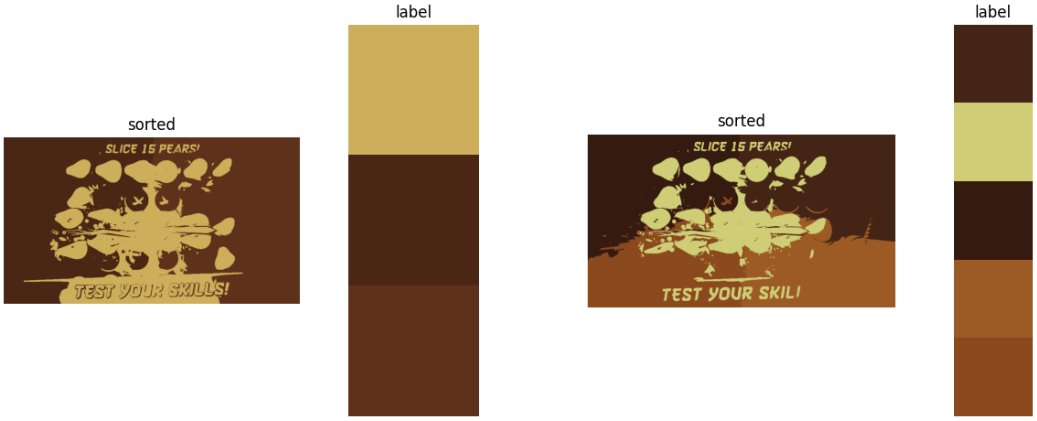
다음은 게임 이미지의 YUV 값을 이용하여 k-means 알고리즘을 통해 분류한 결과이다.



YUV 값을 이용한 결과, k=3의 경우에는 RGB값을 이용한 결과와 유사하게 분류되었다. k=5의 경우 RGB에서 분류된 것과는 다르게 과일의 그림자와 폭탄의 테두리와 X 글자가 다른 그룹으로 분류되어 매우 높은 성능을 확인할 수 있었다. 이는 YUV 영역에서 두 색상의 차이가 RGB 영역에서의 차이보다 크기 때문이라고 해석할 수 있다.

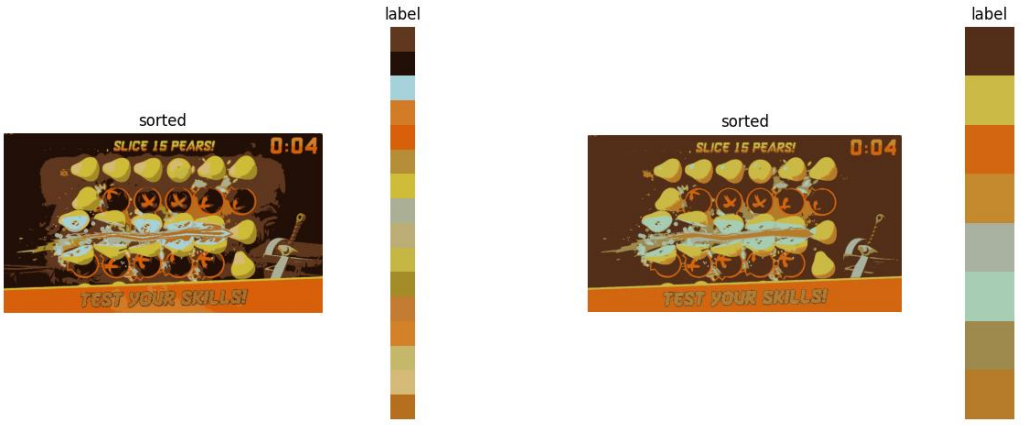
1.5.a.4) K-means with (Y', U, V, Y, X)

다음은 게임 이미지의 YUV 값과 YX 값을 이용하여 k-means 알고리즘을 통해 분류한 결과이다. 다음 사진을 보면 이 경우에는 분류가 잘 되지 않은 것을 확인할 수 있었다.



1.5.b.1) Mean shift with (R, G, B)

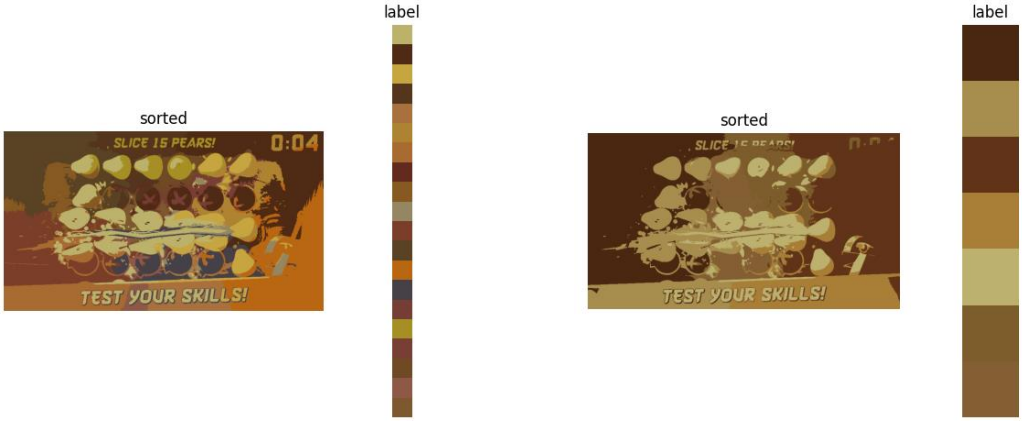
다음은 게임 이미지의 RGB 값을 이용하여 mean shift 알고리즘(h=60, h=70)을 통해 분류한 결과이다.



h=60의 경우 16개의 색으로 분류되었으며, 과일과 과일의 속살, 배경, 폭탄의 테두리, 과일의 그림자 등으로 분류된 것을 확인할 수 있었다. h=70의 경우 8개의 색상으로 분류되었으며, 배경이 모두 한 색으로 분류되었고, 폭탄의 테두리, 과일, 과일의 속살 등으로 분류되어서 분류 성능이 k-means를 적용하였을 때보다 성능이 높은 것을 확인할 수 있었다.

1.5.b.2) Mean shift with (R, G, B, Y, X)

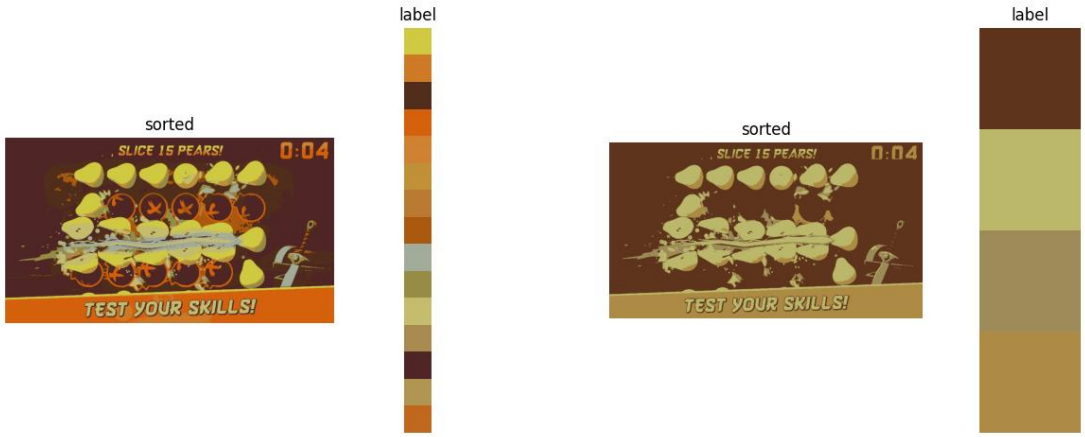
다음은 게임 이미지의 RGB 값과 YX 값을 이용하여 mean shift 알고리즘을 통해 분류한 결과이다.



RGBYX를 이용하였을 경우 분류가 잘 되지 않은 것을 확인할 수 있었다. 이는 개체가 적당한 간격으로 떨어져 있어 위치 정보를 이용하면 색상 정보의 영향이 떨어지기 때문이라고 생각할 수 있다.

1.5.b.3) Mean shift with (Y, U, V)

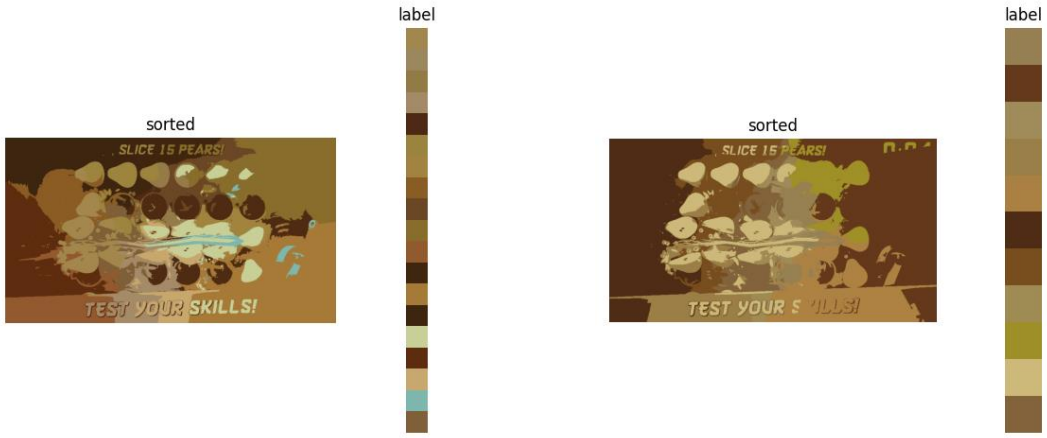
다음은 게임 이미지의 YUV 값을 이용하여 mean shift 알고리즘(h=40, h=50)을 통해 분류한 결과이다.



h=40의 경우 15개의 그룹으로 분류되었으며, 배경과 배경의 칼, 과일의 밝은 부분과 어두운 부분, 폭탄의 테두리가 적절하게 분류된 것을 확인할 수 있었다. h=50의 경우 4개의 그룹으로 분류되었으며, 배경과 과일이 잘 분류된 것은 확인할 수 있었지만 폭탄은 제대로 분류되지 않아 배경과 같은 그룹으로 분류되었다.

1.5.b.4) Mean shift with (Y', U, V, Y, X)

다음은 게임 이미지의 YUV 값과 YX 값을 이용하여 mean shift 알고리즘을 통해 분류한 결과이다.



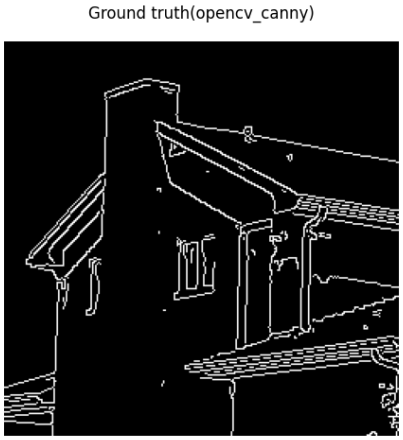
Y'UVYX를 이용하였을 경우엔 RGBYX를 이용하였을 경우와 마찬가지로 분류가 잘 되지 않은 것을 확인할 수 있었다. 이는 마찬가지로 위치 정보를 이용하여 색상 정보의 영향이 떨어지기 때문이다.

분류 알고리즘을 사용할 때 입력하는 정보에는 색상 정보와 위치 정보가 있다. 색상 정보는 색 공간을 변환하여 입력할 수 있고, 위치 정보를 적용할 때에는 적당한 scaling을 하여 위치 정보의 영향을 조절할 수 있다. 하지만 적절하지 않은 색 공간을 사용하거나 적절하지 않은 상황에 위치 정보를 추가하면 오히려 역효과가 날 수 있으므로 적절하게 선택하는 것이 중요하다.

2. Design an edge detector using machine learning technology.

2.1) Use a built-in canny edge detector to obtain edge images.

OpenCV 라이브러리를 사용하여 영상의 canny edge를 검출한 결과는 다음과 같다. Canny edge를 검출할 때 threshold는 100, 150을 사용하여 검출하였다. 다음 영상은 앞으로 적용할 필터와 머신 러닝 기술의 성능을 평가할 때 기준으로 사용하였다.



2.2) Apply Prewitt and Sobel filters to check the performance. (Precision and recall values, and F1-score)

원본 이미지를 Prewitt filter와 Sobel filter를 적용하는 함수를 작성하였다. Filter를 적용할 때에는 이전 프로젝트에서 사용한 apply_filter2 함수를 사용하였다. apply_filter2 함수는 행렬을 mask의 크기만큼 확장한 후 mask의 요소를 하나씩 적용시켜 3번째 차원으로 적층한 후 3번째 차원의 방향으로 합치는 함수로, 각각 픽셀의 sub_image를 사용하여 계산하는 방법보다 큰 속도 향상을 얻을 수 있었던 함수이다. 각각의 filter의 mask는 다음과 같다.

$$m_{y_{prewitt}} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}, \quad m_{x_{prewitt}} = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, \quad m_{y_{sobel}} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}, \quad m_{x_{sobel}} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix},$$

y 방향과 x 방향의 필터를 적용한 후 이를 더하여 edge를 검출하는 필터로, 필터링 후 나온 영상에 binary_image 함수를 작성하여 임계값 이상의 값을 255로 적용시켰다. 그 후 필터의 성능을 평가하지 위하여 여러 성능 지표(precision, recall, f1 score, FPR, TPR, confusion matrix)를 반환하는 performance 함수까지 작성하였다.

$$precision = \frac{TP}{TP + FP}, \quad recall = \frac{TP}{TP + FN}, \quad FPR = \frac{FP}{FP + TN}, \quad TPR = \frac{TP}{TP + FN}$$

$$f1\ score = \frac{2}{\frac{1}{recall} + \frac{1}{precision}} = 2 \times \frac{precision \times recall}{precision + recall}, \quad confusion\ matrix = \begin{bmatrix} TP & FP \\ FN & TN \end{bmatrix}$$

Precision은 정밀도로 참으로 검출한 값 중 정답의 비율을 표현한 값이고, recall은 재현율로 실체 참 값 중 검출한 참값의 비율을 뜻한다. FPR은 실제 거짓 값 중 참으로 예측한 값이다. TPR은 recall과 같다. FPR을 제외하고 나머지는 모두 높을수록 좋은 성능을 가진다고 할 수 있다.

작성된 주요 함수는 다음과 같다.

```
def binary_image(img, th, v):
    binary = np.zeros_like(img)
    binary[img>=th] = v
    return binary
```

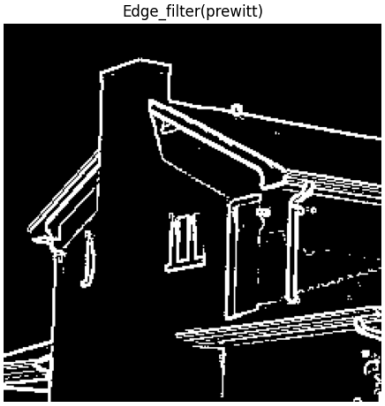
```
def apply_filter2(img, fil):
    h1 = fil.shape[0]//2
    h2 = fil.shape[1]//2
    result = np.zeros((img.shape[0]+2*h1, img.shape[1]+2*h2))
    ar1 = np.arange(-h1,h1+1)
    ar2 = np.arange(-h2,h2+1)
    for y in ar1:
        for x in ar2:
            sub = np.pad(img,((h1-y,h1+y),(h2-x,h2+x)),'edge')
            result += fil[y+h1,x+h2] * sub
    return result[h1:-h1,h2:-h2]
```

```
def edge_filter(img, fil, th=30, plot=0):
    if fil=='prewitt':
        fil_y = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])/6
        fil_x = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])/6
    if fil=='sobel':
        fil_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])/8
        fil_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])/8
    edge_y = apply_filter2(gray, fil_y)
    edge_x = apply_filter2(gray, fil_x)
    edge = edge_y + edge_x
    edge = binary_image(edge, th, 255)
    return edge
```

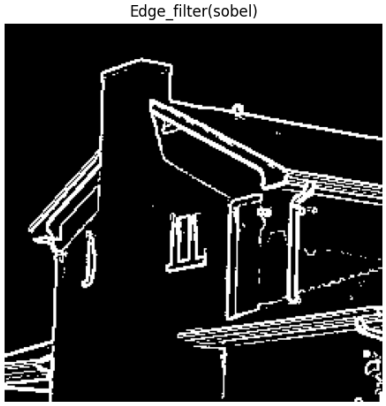
```
def performance(truth, model):
    truth = truth.astype(bool)
    model = model.astype(bool)
    TP = np.sum(truth * model)
    FP = np.sum(np.invert(truth) * model)
    FN = np.sum(truth * np.invert(model))
    TN = np.sum(np.invert(truth) * np.invert(model))
    confusion_mat = np.array([[TN, FP],[FN, TP]])
    precision = TP/(TP+FP)
    recall = TP/(TP+FN)
    accuracy = (TP+TN)/(TP+FN+FP+TN)
    f1 = 2*(precision*recall)/(precision+recall)
    FPR = FP/(TN+FP)
    TPR = TP/(TP+FN)
    return [precision, recall, f1, FPR, TPR, confusion_mat]
```

두 filter를 통해 검출된 edge와 측정된 filter의 성능은 다음과 같다.

Edge_filter(prewitt)



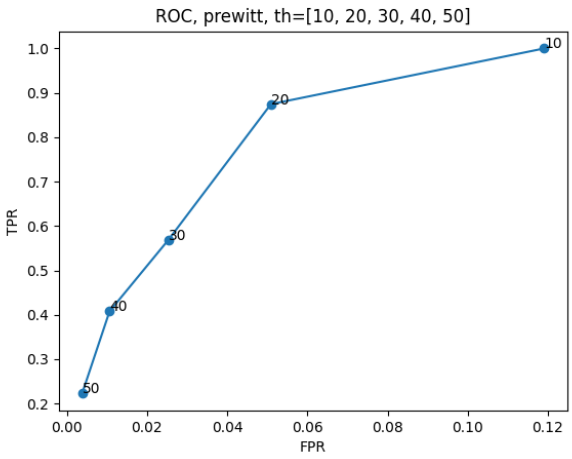
Edge_filter(sobel)

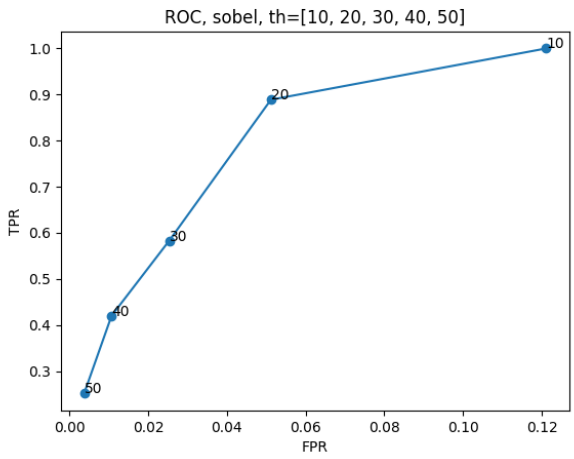


```
Prewitt : (precision, recall, f1)=(0.5245, 0.8743, 0.6557)
Sobel : (precision, recall, f1)=(0.5273, 0.8887, 0.6619)
```

두 filter 모두 recall 값은 0.87~0.89의 값을 가지고 precision 값은 0.52~0.53의 값을 가지는 것을 확인할 수 있었다. F1 score의 경우 precision과 recall의 조화평균으로 0.66의 값을 가졌다. 이를 통해 두 filter는 실제 edge는 거의 검출하지만 검출한 edge는 절반 정도의 확률로 실제 edge라는 것을 알 수 있다. 이는 canny edge로 검출한 edge는 얇은 선을 가지는 데 비해 두 filter는 edge를 두껍게 검출하기 때문이라는 것을 예상할 수 있는데, 실제 검출된 edge 이미지를 살펴보면, canny edge의 경우 edge의 두께가 1인 것에 비해 두 filter의 edge는 두께가 2인 것을 확인할 수 있었다.

Filtering 된 영상을 threshold를 통해 이진화하는 과정에서 threshold 값을 결정하게 되는데 threshold 값을 통해 ROC curve를 그릴 수 있었다. 각각 filter에 대해 threshold=[10, 20, 30, 40, 50]에 따른 ROC curve는 다음과 같다.





ROC curve를 보면 TPR을 증가시키기 위해 threshold 값을 낮추게 되면 FPR 또한 높아지게 되어 두 값 간의 trade-off가 필요하다. 두 filter의 경우 threshold 값을 10으로 설정할 경우 거의 모든 edge를 검출하게 되지만 검출된 edge 중 12%는 실제 edge가 아니게 됨을 알 수 있다. ROC curve는 곡선 아래의 면적 AUC가 클수록 성능이 좋다.

2.3) Use the following machine learning schemes. Evaluate the performance.

Scikit-learn 라이브러리를 이용하여 픽셀 주위의 $K \times K$ 픽셀 값에 대한 machine learning 기법을 사용하여 edge를 검출하였다. 분류기에 입력하는 X 값은 는 sub_image 함수를 작성하여, (픽셀의 개수 × sub image의 크기)의 크기를 가지는 벡터로 변경하였고, y 값 또한 위에서 작성한 binary_image 함수를 이용하여 ground truth 이미지를 0과 1로 이진화 시켰다.

```
def sub_image(img, K, vector=0):
    sub = np.zeros(img.shape+(K**2,))
    k = K//2
    img = np.pad(img, k, 'edge')
    for i in range(K):
        for j in range(K):
            sub[:,:,K*i+j] = img[i:img.shape[0]-2*k+i, j:img.shape[1]-2*k+j]
    if vector==1: sub = sub.reshape([img.size,K**2])
    return sub
```

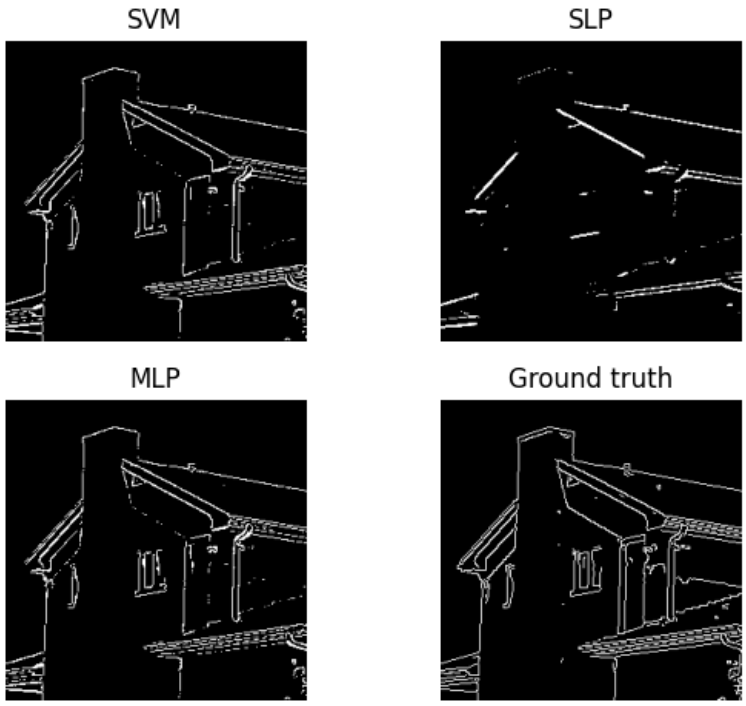
2.3.a~c) SVM, SLP, MLP

SVM, SLP, MLP로 edge를 검출한 결과는 다음과 같다. Sub-image의 크기는 3×3, SVM의 경우 rbf 커널을 사용하였고, MLP의 경우 10의 크기를 가지는 1개의 hidden layer를 사용하였다. Training set로 학습한 후 test set으로 edge를 검출한 결과 성능은 다음과 같다.

```
SVM : (precision, recall, f1)=(0.8526, 0.6526, 0.7393), 15.423sec
SLP : (precision, recall, f1)=(0.6161, 0.2305, 0.3355), 0.043sec
MLP : (precision, recall, f1)=(0.8382, 0.6518, 0.7333), 11.082sec
```

성능 지표의 경우 SVM과 MLP는 precision은 0.84~0.85, recall은 0.65정도로 Prewitt, Sobel filter보다 precision은 높았으며, recall은 작다. 이는 검출된 edge 중 실제 edge인 경우는 많았으나, 실제 edge 중 검출된 edge는 filter보다 조금 성능이 낮다는 것을 알 수 있다. SLP의 경우 세 방법 중 압도적으로 학습 시간이 짧았으나 성능은 나머지 두 방법보다 나빴다. 특히 recall의 경우 0.23으로 실제 edge를 잘 검출하지 못하는 것을 알 수 있다.

Training set으로 학습한 후 다시 모든 dataset을 넣어 edge 이미지를 얻은 결과는 다음과 같다.



검출된 edge의 영상을 보면 SVM과 MLP의 경우 육안으로도 잘 검출된 것을 확인할 수 있었다. SLP의 경우 영상에서는 y 방향의 edge는 잘 검출하였지만 x 방향의 edge는 거의 검출하지 못한 것으로 보였고, 반복 실행해 본 결과 x, y 방향에 관계없이 한쪽 방향의 edge를 잘 검출하지 못하는 것을 확인할 수 있다.

2.4) Compare the performance of various machine learning conditions.

머신 러닝 기법을 사용할 때, 입력하는 sub-image의 크기와 training dataset의 비율을 조절하여 최적의 검출 방법을 찾을 수 있다. 또한 SVM을 이용할 경우 사용하는 커널에 따라 더 정확한 결과를 가질 수 있으며, MLP의 경우 hidden layer의 크기와 개수에 따라 성능을 향상시키거나 학습 속도를 향상시킬 수 있다.

2.4.a) Sub-image size(K)

먼저 입력하는 sub-image의 크기(K=3, K=5, K=7)에 따른 edge 검출 결과를 살펴보면 다음과 같다.

```
SVM : (precision, recall, f1)=(0.8526, 0.6526, 0.7393), 15.423sec
SLP : (precision, recall, f1)=(0.6161, 0.2305, 0.3355), 0.043sec
MLP : (precision, recall, f1)=(0.8382, 0.6518, 0.7333), 11.082sec

SVM(k=5) : (precision, recall, f1)=(0.8719, 0.6608, 0.7518), 15.170sec
SLP(k=5) : (precision, recall, f1)=(0.8889, 0.0135, 0.0266), 0.030sec
MLP(k=5) : (precision, recall, f1)=(0.8352, 0.6414, 0.7255), 9.839sec

SVM(k=7) : (precision, recall, f1)=(0.8355, 0.6544, 0.7340), 15.161sec
SLP(k=7) : (precision, recall, f1)=(0.3783, 0.1340, 0.1979), 0.066sec
MLP(k=7) : (precision, recall, f1)=(0.8069, 0.6954, 0.7470), 14.048sec
```

세 방법 모두 입력하는 k에 따라 성능이나 계산시간의 변화가 일관적이지 않은 것을 확인할 수 있었다. 따라서 해당 영상에서 edge를 검출할 경우에는 굳이 큰 sub-image 벡터를 입력하지 않고 3×3정도만 입력하여도 된다고 판단된다.

2.4.b) Training data size

다른 방법은 training data의 크기를 변화시키는 방법이다. 다음은 70%, 60%, 50%의 training data를 입력하였을 때 학습한 결과이다.

```
SVM : (precision, recall, f1)=(0.8526, 0.6526, 0.7393), 15.423sec
SLP : (precision, recall, f1)=(0.6161, 0.2305, 0.3355), 0.043sec
MLP : (precision, recall, f1)=(0.8382, 0.6518, 0.7333), 11.082sec

SVM(train=60%) : (precision, recall, f1)=(0.8430, 0.6813, 0.7536), 14.659sec
SLP(train=60%) : (precision, recall, f1)=(0.6762, 0.0457, 0.0856), 0.030sec
MLP(tran=60%) : (precision, recall, f1)=(0.8117, 0.7160, 0.7609), 10.345sec

SVM(train=50%) : (precision, recall, f1)=(0.8591, 0.6356, 0.7306), 13.275sec
SLP(train=50%) : (precision, recall, f1)=(0.2570, 0.1434, 0.1840), 0.026sec
MLP(train=50%) : (precision, recall, f1)=(0.8317, 0.7058, 0.7636), 11.145sec
```

SVM과 MLP의 경우 training dataset의 비율이 줄었을 때의 계산속도 향상에 비해 정확도는 거의 일정한 것을 확인할 수 있었다. 따라서 edge 검출에서 SVM과 MLP를 사용할 때에는 큰 training set을 사용할 필요가 없다는 것을 알 수 있다. SLP의 경우 일관적이지 않은 성능을 보여주고 있으며, SLP는 강력하거나 강건한 머신 러닝 기법이 아니라고 생각할 수 있다.

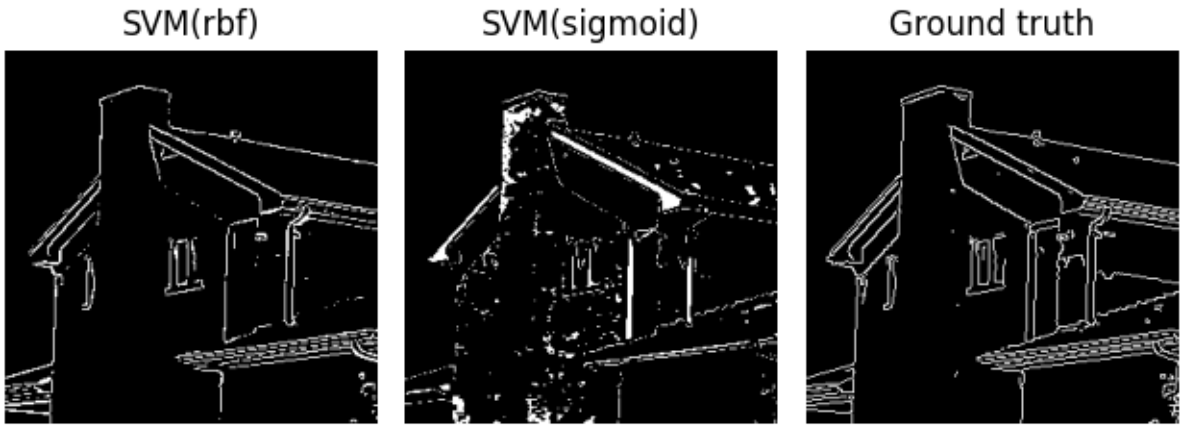
2.4.c) SVM kernel

SVM kernel에 따라 edge 검출 성능을 살펴보았다.

```
SVM(rbf) : (precision, recall, f1)=(0.8526, 0.6526, 0.7393), 15.423sec
SVM(sigmoid) : (precision, recall, f1)=(0.0988, 0.0836, 0.0906), 16.919sec
```

Sigmoid 커널을 이용할 경우 precision과 recall 성능 모두 떨어진 것을 확인할 수 있었다. 해당 영상에는 edge의 곡률이 거의 없어 곡선을 검출할 때 주로 사용하는 sigmoid 커널의 경우 큰 성능을 기대하기는 어려웠다.

다음은 각각 커널을 사용하여 학습한 후 모든 데이터를 다시 사용하여 edge를 검출한 결과이다.



육안으로도 rbf 커널을 사용할 때에는 ground truth 영상과 거의 유사하지만, sigmoid 커널을 사용할 때, 위양성이 많이 검출되어, edge 검출이 잘 이루어지지 않은 것을 확인할 수 있었다.

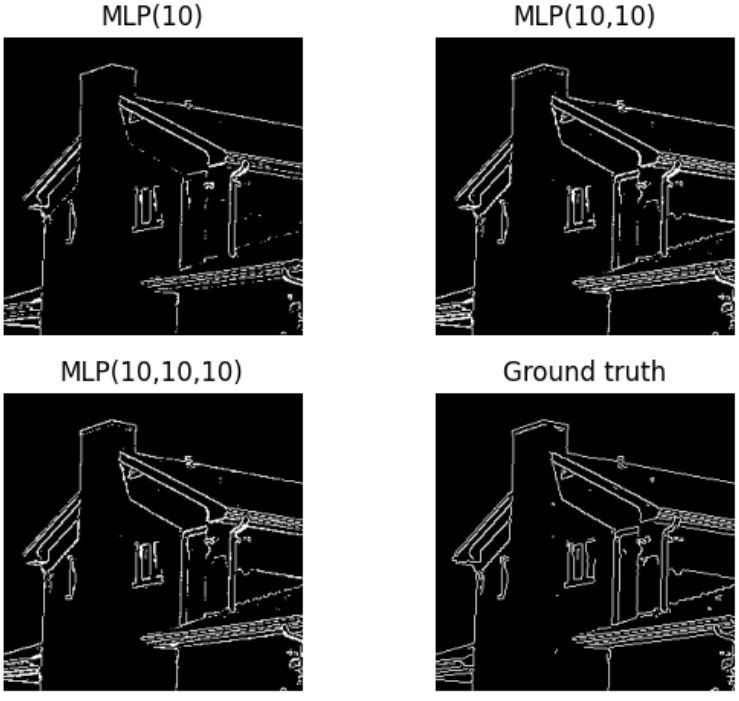
2.4.d) MLP layer

MLP 기법을 사용할 때에는 MLP의 hidden layer의 크기와 개수에 따라 성능을 개선할 수 있다. Hidden layer의 개수에 따른 MLP의 성능 변화를 알아보기 위해 hidden layer의 크기는 각각 개수가 1, 2, 3일 때의 결과를 확인하면 다음과 같다.

```
MLP(10) : (precision, recall, f1)=(0.8382, 0.6518, 0.7333), 11.082sec
MLP(10,10) : (precision, recall, f1)=(0.8034, 0.8425, 0.8225), 18.860sec
MLP(10,10,10) : (precision, recall, f1)=(0.8316, 0.8214, 0.8265), 23.935sec
```

Hidden layer의 개수가 늘어남에 따라 학습 시간은 크게 증가하였으며, precision의 경우 layer가 2겹일 때 성능이 소폭 줄어들었으나, 3겹일 때 다시 늘어났다. Recall의 경우 layer가 2겹일 때 크게 증가하였으며, 3겹일 때에는 크게 증가하지 않았다. 따라서 계산 시간이 중요할 경우 1겹의 layer로도 충분하다고 생각할 수 있으며, 정확도가 중요할 경우 2겹의 layer를 사용하는 것이 최적이라고 판단할 수 있다.

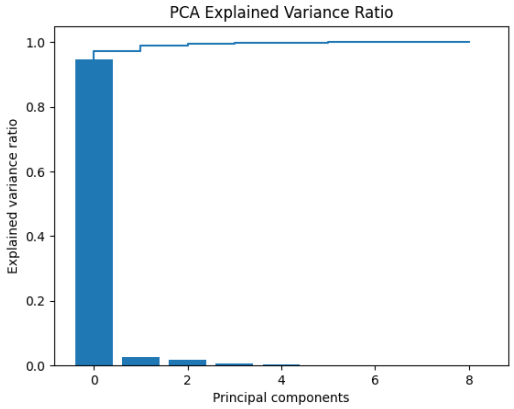
다음은 학습 후 다시 모든 data에 대해 분류를 진행하여 edge image를 만든 결과이다.



Edge 영상을 보면 세 영상 모두 준수하게 edge를 검출한 것을 알 수 있었다. 1겹을 사용하였을 때에 검출하지 못한 edge가 2겹 이상을 사용하였을 때에는 검출한 부분이 많으며, 3겹을 사용하였을 때는 2겹을 사용하였을 때에 비해 큰 성능향상이 없는 것을 확인할 수 있었다.

2.4.e) Dimension reduction using PCA

PCA는 각 특성(주위 픽셀값)의 선형조합을 통해 잠재 변수를 도출한다. 도출된 각각의 잠재변수의 explained variance를 측정하여 explained variance가 높은 변수만을 채택하여 차원을 축소하여 복잡성을 줄이는 것이 목표이다. Scikit-learn 라이브러리를 사용하여 표준화된 train data의 explained variance를 살펴보면 오른쪽 그래프와 같다.



99%의 explained variance를 사용하면 9개(K=3)의 특성에서 4개의 잠재 변수만을 사용하여 계산할 수 있었다.

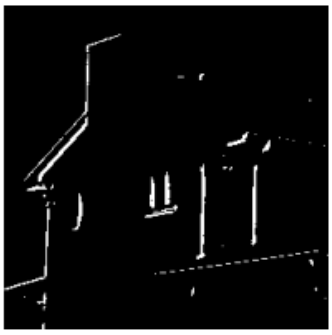
```
SVM : (precision, recall, f1)=(0.8526, 0.6526, 0.7393), 15.423sec
SVM with PCA : (precision, recall, f1)=(0.8327, 0.6867, 0.7527), 12.748sec

SLP : (precision, recall, f1)=(0.6161, 0.2305, 0.3355), 0.043sec
SLP with PCA : (precision, recall, f1)=(0.5188, 0.1899, 0.2781), 0.035sec

MLP : (precision, recall, f1)=(0.8382, 0.6518, 0.7333), 11.082sec
MLP with PCA : (precision, recall, f1)=(0.7829, 0.7435, 0.7627), 11.127sec
```

SVM의 경우 PCA로 차원 축소한 후 학습한 결과 학습 시간이 유의미하게 줄은 것을 확인할 수 있었으며, 성능 하락은 거의 없었다. SLP의 경우 학습 시간은 줄었으나 분류 성능도 하락했다. MLP의 경우 학습 시간은 유사하였으며, 분류 성능은 소폭 상승한 것을 확인할 수 있었다.

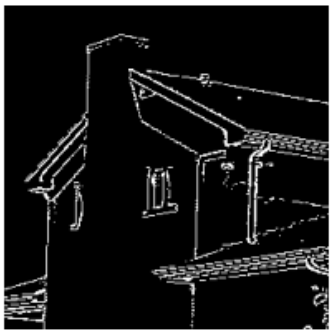
다음은 학습된 결과로 검출한 edge 영상이다.

SVM with PCA

SLP with PCA



MLP with PCA

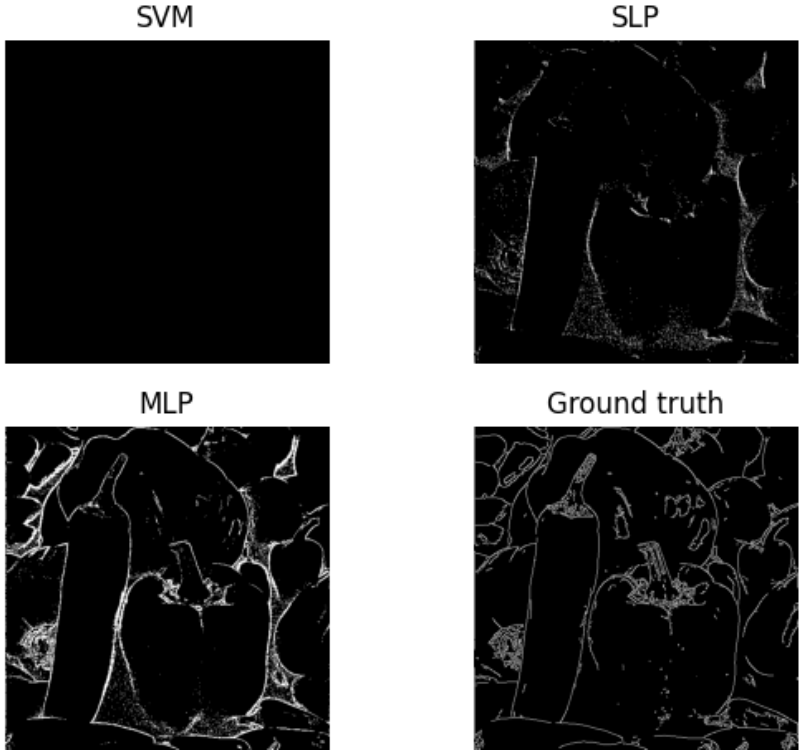


Ground truth

PCA를 사용한 영상과 사용하지 않은 2.3)의 영상과 비교하였을 때, SVM과 MLP의 경우 큰 차이가 없었으며, SLP의 경우 PCA를 사용하지 않았을 때와 마찬가지로 한쪽 방향의 edge가 잘 검출되지 않는 것을 확인할 수 있었다.

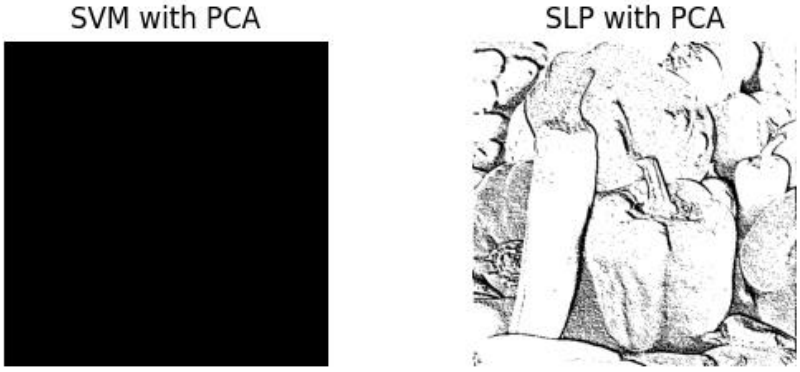
2.4.f) Apply on other images.

house.bmp의 canny edge로 2.3)과 2.4.e)에서 학습된 분류기로 다른 이미지의 edge를 잘 검출할 수 있는지 확인하기 위하여 pepper.bmp와 lena.bmp의 edge를 검출해 보았다. Ground truth 영상은 canny edge를 검출한 영상이다.

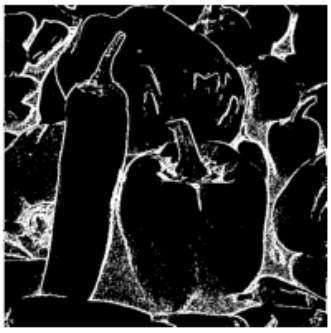


SVM의 경우 edge를 검출하지 못하였고, SLP는 house.bmp의 경우와 마찬가지로, 한쪽 방향의 edge를 검출하지 못하였다. 반복 실행한 결과, 어떤 학습 dataset(house.bmp)에서는 반전된 edge map을 보여주기도 하였다. MLP의 경우 edge를 어느정도 잘 검출한 것을 확인할 수 있었다. SVM의 경우 영상이 바뀌게 되면 믿을 수 없으며, SLP의 경우 여러 변수가 바뀌게 되면 일관적이지 않은 결과를 보여준다는 것을 이전 다른 항목에서 많이 확인할 수 있었다. MLP는 영상이 변화하여도 edge를 잘 검출하여, edge detection에서는 MLP를 사용하는 것이 현명해 보인다.

PCA로 차원을 축소한 후 검출된 edge 영상은 다음과 같다.



MLP with PCA



Ground truth

SVM의 경우 PCA를 사용하지 않았을 때와 마찬가지로 edge를 검출하지 못하였으며, SLP의 경우 한쪽 방향의 edge를 검출하지 못하거나, edge map의 반전된 영상을 보여주는 등, 일관적이지 않았다. MLP의 경우 edge를 잘 검출하는 것을 확인할 수 있었다.

```
SVM predict time = 136.833      SVM_PCA predict time = 128.008
SLP predict time = 0.010        SLP_PCA predict time = 0.007
MLP predict time = 0.065        MLP_PCA predict time = 0.106
```

학습시간과 별개로 예측 시간 또한 SVM의 경우 예측할 데이터가 많아지면 예측 시간이 크게 증가하였으며, SLP와 MLP는 예측 시간은 매우 작은 것을 확인할 수 있었다. MLP의 경우는 PCA를 사용함에 따라서 학습 시간이나, 예측 시간이 유의미하게 증가하지 않았다. 학습 시간이 충분할 경우 PCA를 적용하지 않은 MLP를 사용하는 것이 적절하다고 판단된다.