1. (K-means/Mean shift) Implement the followings, and analyze and compare the results.

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

2) Cluster the image based on the (R, G, B) vector of the image, and visualize it as below

<k-means>

(1) feature range, feature dimension을 갖는 벡터를 랜덤으로 생성해서 centroid를 초기화한다. 이미지의 RGB값을 feature로 사용했기 때문에 range는 0~255, dimension은 3차원으로 centroids를 초기화 했다.

```
centroid = np.random.uniform(0,255,(k,3)).astype(np.uint8) # random initialization
print('start')
print(centroid)
```

(2) 초기화한 centroids와 각 픽셀 값 사이의 거리를 측정한 뒤 가장 가까운 centroid의 group에 각 픽셀 값과 그 위치를 할당해서 k개의 group을 생성한다.

```
# groups: 각 그룹의 RGB값 저장
groups = []
# indexes: 각 그룹의 index 저장
indexes = []
# cluster 개수에 맞게 리스트 추가
for d in range(k):
    groups.append([])
    indexes.append([])

# 거리가 가장 가까운 group에 각 RGB값과 index 추가
for i in range(h):
    for j in range(w):
        distances = np.array([])
        for d in range(k):
            distances = np.append(distances, np.sum((img[i,j,:] - centroid[d])**2))
        idx=np.argmin(distances)
        groups[idx].append(img[i,j])
        indexes[idx].append([i,j])
```

(3) 각 그룹에 속하는 픽셀 값들의 평균값을 계산해 다음 centroid를 계산한다.

```
# 각 그룹의 픽셀 평균값으로 다음 centroid 계산
next_centroid = np.zeros((k,3),dtype='int')
for d in range(k):
    next_centroid[d] = np.array(groups[d]).mean(axis=0)
print(next_centroid)
```

(4) 종료조건으로, 이전 centroid와 다음 centroid가 같은 지 비교한다. 같은 경우 반복을 종료하고 같지 않은 경우 centroid 위치를 변경한 뒤에 위 (2), (3)번 과정을 종료조건 만족시까지 반복한다.

```
# 종료조건
if(np.array_equal(centroid, next_centroid)):
    print('find!')
    break
centroid = next_centroid
```

(5) 반복이 끝나서 centroid가 결정되면 저장해 두었던 각 group의 index를 사용해 각 그룹의 centroid값으로 각 이미지 픽셀들을 대체한다.

```
# Cluster the image: 각 그룹에 centroid 값 할당
for i in range(k):
    for j in range(len(indexes[i])):
        img[indexes[i][j][0],indexes[i][j][1],:]=centroid[i]
```

<k-means 결과 출력/분석>

k-means의 경우 k값을 parameter로 사람이 직접 입력해줘야 한다는 단점이 있다. 비교 분석을 위해 k값을 5와 9로 설정해서 결과를 확인했다. 추가로 centroids initialization에 따라 결과에 영향을 미치므로 2번 실행해 결과를 확인했고 outlier에 약한 특징을 확인하기 위해 k=3인 경우 이미지에 대한 출력을 확인했다

- centroid 변화 출력

다음은 centroids가 변화하는 모습을 확인할 수 있는 출력화면이다. K=9인 경우 centroids가 수렴하는데 더 시간이 오래 걸렸다. 이는 거리 측정 시간이 증가했기 때문이다. 반복이 진행됨에 따라 각 centroid의 변화량이 점점 줄어들며 수렴하는 것을 확인할 수 있었다.

k=5

```
start
[[129  22  47]
 [  1  36  25]
 [156 128  17]
 [169  46 145]
 [ 96 192 217]]
number:  1
[[139 124 132]
 [140 124 132]
 [149 137 147]
 [150 112 112]
 [148 146 160]]
number:  2
[[119 122 127]
 [156  37 195]
 [149 145 139]
 [136  89  90]
 [161 196 215]]
number:  5
[[119 105 114]
 [159   0 222]
 [166 120 112]
 [127  80  83]
 [162 198 218]]
number:  6
[[120 105 113]
 [159   0 222]
 [166 109 100]
 [101  64  76]
 [162 197 217]]
number:  7
[[122 106 113]
 [159   0 222]
 [166 106  97]
 [ 93  58  73]
 [162 197 217]]
number:  8
[[122 106 112]
 [159   0 222]
 [166 106  96]
 [ 93  57  73]
 [162 197 217]]
number:  9
[[121 106 112]
 [159   0 222]
 [166 106  96]
 [ 92  57  73]
 [162 197 217]]
number:  10
[[121 106 112]
 [159   0 222]
 [166 106  96]
 [ 92  57  73]
 [162 197 217]]
find!
```

k=9

```
start
[[140  15 161]
 [156  64  26]
 [ 79 245  88]
 [120 198 149]
 [204  71  52]
 [ 44 193 113]
 [ 98 198 116]
 [ 90  81 239]
 [162 180  21]]
number:  1
[[146 121 124]
 [151 161 180]
 [146 117 119]
 [138 123 133]
 [139 127 136]
 [149 119 122]
 [144 134 144]
 [142 120 125]
 [148 127 131]]
number:  2
[[143 134 113]
 [162 197 216]
 [132  86  88]
 [130  78 160]
 [124 134 137]
 [173 115 106]
 [137 146 149]
 [115 108 114]
 [166 142 124]]
number:  14
[[119  96 103]
 [163 199 220]
 [ 75  41  64]
 [159   0 222]
 [127 136 140]
 [166 104  94]
 [148 167 173]
 [103  66  78]
 [172 125 115]]
number:  15
[[119  95 102]
 [163 199 220]
 [ 73  40  64]
 [159   0 222]
 [127 135 140]
 [165 104  94]
 [148 167 173]
 [102  65  77]
 [172 120 111]]
number:  16
[[119  95 102]
 [163 199 220]
 [ 72  39  63]
 [159   0 222]
 [127 135 139]
 [165 103  93]
 [149 167 173]
 [101  64  76]
 [172 117 108]]
number:  23
[[118  94 101]
 [163 199 220]
 [ 70  37  63]
 [159   0 222]
 [128 135 138]
 [162 100  90]
 [149 166 172]
 [100  63  76]
 [170 111 102]]
number:  24
[[118  94 101]
 [163 199 220]
 [ 70  37  63]
 [159   0 222]
 [128 135 138]
 [162 100  90]
 [149 166 172]
 [100  63  76]
 [170 111 101]]
number:  25
[[118  94 101]
 [163 199 220]
 [ 70  37  63]
 [159   0 222]
 [128 135 138]
 [162 100  90]
 [149 166 172]
 [100  63  76]
 [170 111 101]]
find!
```

 # 두번째 실행: 두 번째 실행결과 큰 차이점으로 반복횟수 및 최종 centroids가 바뀐 것을 확인할 수 있었다. k=5인 경우 12번의 반복을 했고 k=9인 경우 29번만에 최종 출력을 했다. 결과값 또한 달라진 것을 확인할 수 있었다.

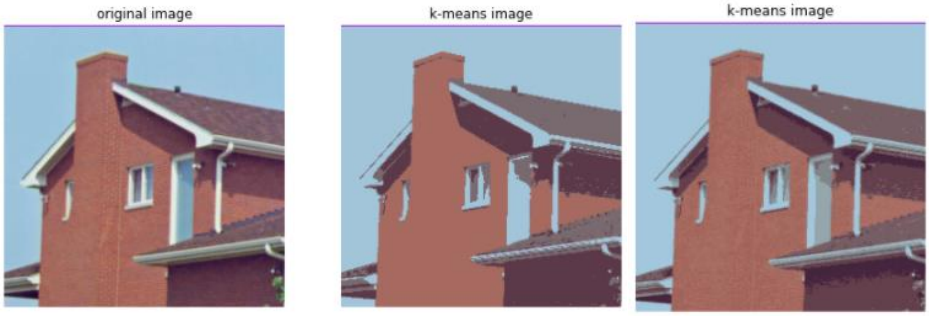
```
< result centroid >
[[ 96  63  77]
 [159   0 222]
 [163 199 219]
 [159 105  98]
 [137 151 156]]
```

```
< result centroid >
[[119  94 101]
 [159   0 222]
 [157 197 220]
 [ 70  37  63]
 [149 166 172]
 [166 106  96]
 [100  63  76]
 [129 135 137]
 [212 221 215]]
```

- 이미지 출력

house이미지를 사용해 clustering을 진행했다. 첫번째 이미지는 원본 이미지, 두번째 이미지는 k=5인 경우, 세번째 이미지는 k=9인경우 결과 이미지이다. 유사한 색을 가지는 영역끼리 같은 색을 가지도록 영역 분할이 된 것을 확인할 수 있었는데 이미지를 확인해보면 parameter로 넣어준 k값만큼 영역분할이 된 것을 확인할 수 있었다. 따라서 k=9에서 영역분할이 보다 세세하게 된 것을 볼 수 있다.

첫번째 실행



두번째 실행

centroids가 달라졌다. 따라서 영역분할도 첫번째 결과와 다르게 된 것을 확인할 수 있었다.

k=3 실행: 원본이미지를 확인해보면 맨 위쪽에 가로줄로 보라색이 삽입된 것을 확인할 수 있는데 다른 픽셀값들과 다른 경향의 값을 가지는 outlier로 생각할 수 있다. K=3으로 k-means를 수행했고 결과 이미지를 확인하면 k값을 증가시켰을 때와 비교해서 결과 이미지의 전체적인 톤이 원본에 비해 많이 바뀐 것을 확인할 수 있다.

<OpenCV 라이브러리>

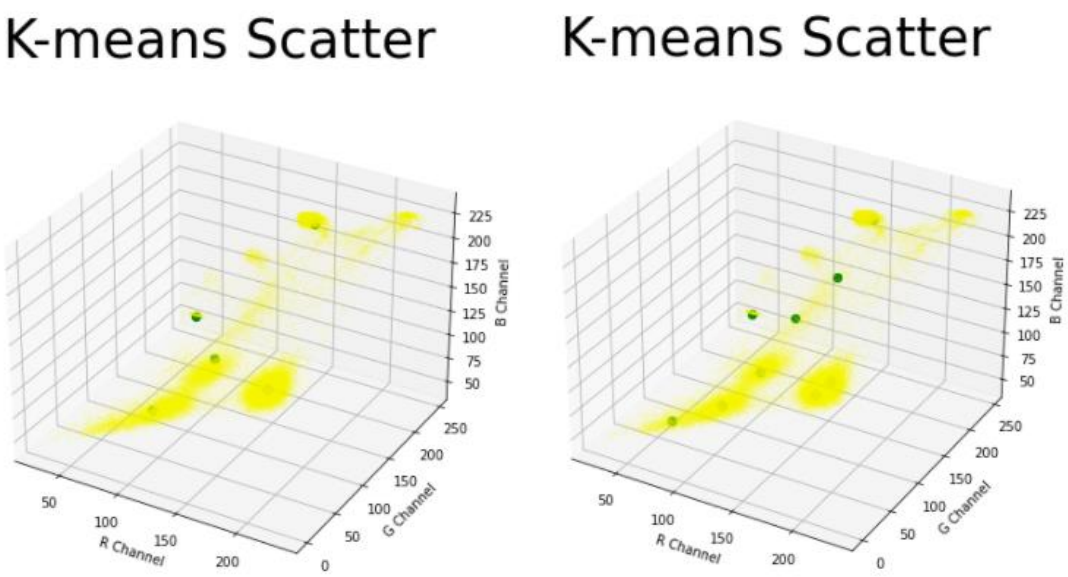
다음은 OpenCV라이브러리를 사용해 k-means를 구현한 내용이다. 유사하게 영역분할이 된 것을 확인할 수 있다.

```
src = cv2.imread('Image/house.bmp')

# 307200, 3
data = src.reshape((-1, 3)).astype(np.float32)

# K-means 알고리즘
criteria = (cv2.TERM_CRITERIA_EPS|cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)

K=5
ret, label, center = cv2.kmeans(data, K, None, criteria, 10, cv2.KMEANS_RANDOM_CENTERS)

center = center.astype('uint8')

dst = center[label.flatten()]
dst = dst.reshape((src.shape))
```

- scattering plot

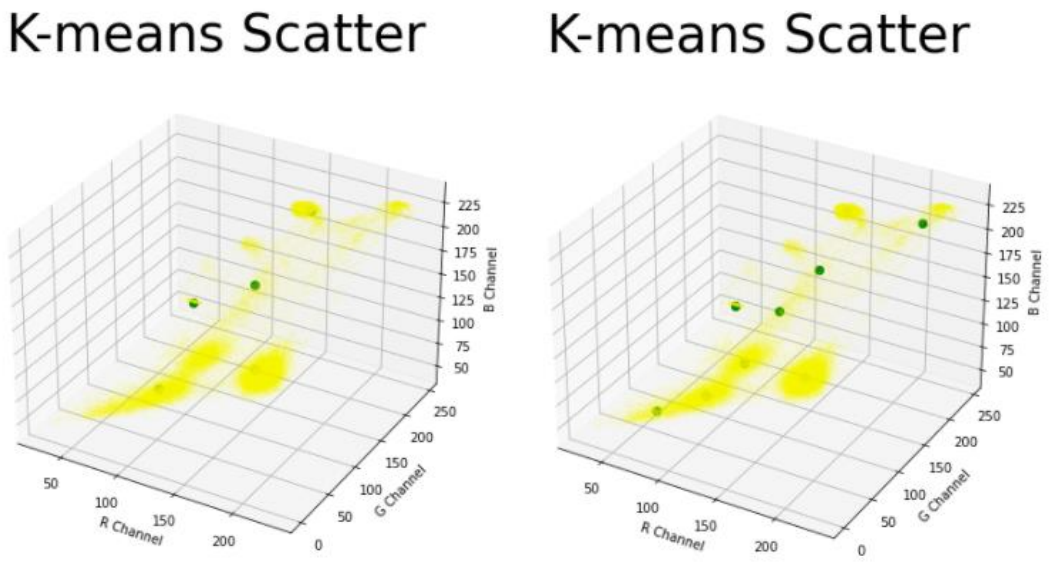
다음은 3차원 그래프로 각 이미지 픽셀과 centroid를 plot한 결과이다. 노란색은 image 픽셀, 초록색은 centroids를 의미한다. 왼쪽은 k=5, 오른쪽은 k=9일 때 결과로 초록색 점의 개수를 확인하면 k값을 직접적으로 확인할 수 있으며 노란색 점들이 많이 뭉친 위치에 centroid가 위치해 의도한 대로 centroid를 찾은 것을 확인할 수 있었다.

 # 첫번째 실행



 # 두번째 실행

같은 지점에 centroid가 위치한 것도 있지만 위치가 달라진 경우도 존재하는 것을 확인할 수 있다. Centroid initialization의 중요성을 다시 한 번 확인할 수 있다.



<mean-shift>

(1) RGB값들을 사용해서 3차원 histogram을 생성하고 0이 아닌 index 값들을 추출한다.

```
# RGB plane에 점들을 나타냄
temp = np.zeros((256,256,256))
for i in range(h):
    for j in range(w):
        r,g,b = img[i,j]
        temp[r,g,b] += 1

# RGB값이 존재하는 index만 가져옴
meanList = []
temp_idx = np.where( temp > 0)
index_R = temp_idx[0]
index_G = temp_idx[1]
index_B = temp_idx[2]
```

(2) 3차원 공간에 대해 masking을 하기 위한 index를 추출한다. Mask 범위가 음수 또는 255를 초과할 수 없으므로 mask의 중심에 따라 mask 공간은 64x64x64 보다 작아질 수 있음에 유의해서 index를 추출했다. 모든 index값(non-zero 픽셀 값)에 대해 shift하며 결과를 출력하면 이미지에 따라 수행시간이 매우 길어지게 되므로 일정 개수(500, 100 두 번 실행)마다 shift를 수행할 수 있도록 했다. 100번으로 수행한 경우 더 많은 개수의 centroid를 반환했다. 수행시간과 영역분할의 정확도 사이에 trade-off 관계가 있음을 예상할 수 있다.

```
mask_size = 32  # 64 x 64 x 64 : 3차원 마스크

# 모든 index값에 대해 실행하면 수행시간 길어짐
for j in range( int(len(index_R)/500)): #len(index_R)):
    i = j*500

    # masking 해야하는 index 추출
    idx1 = np.clip( np.array([index_R[i]-mask_size, index_G[i]-mask_size, index_B[i]-mask_size]), 0, 255)
    idx2 = np.clip( np.array([index_R[i]+mask_size, index_G[i]+mask_size, index_B[i]+mask_size]), 0, 255)
    previous_mean = np.array([index_R[i], index_G[i], index_B[i]])
```

```
mask_size = 32  # 64 x 64 x 64 : 3차원 마스크

# 모든 index값에 대해 실행하면 수행시간 길어짐
for j in range(int(len(index_R)/100) ): #len(index_R)): int(len(index_R)/500)
    i = j*100

    # masking 해야하는 index 추출
    idx1 = np.clip( np.array([index_R[i]-mask_size, index_G[i]-mask_size, index_B[i]-mask_size]), 0, 255)
    idx2 = np.clip( np.array([index_R[i]+mask_size, index_G[i]+mask_size, index_B[i]+mask_size]), 0, 255)
    previous_mean = np.array([index_R[i], index_G[i], index_B[i]])
```

(3) 추출한 mask index에 대해 masking을 수행하고 nonzero index에 대해 픽셀 값과 픽셀 값의 개수를 곱해 누적시킨 후 평균값을 계산한다.

```
while True:
    repeat += 1
    # masking 이후 mean 계산
    subArea = temp[idx1[0]:idx2[0],idx1[1]:idx2[1],idx1[2]:idx2[2]].copy()
    index_r = np.nonzero(subArea)[0]
    index_g = np.nonzero(subArea)[1]
    index_b = np.nonzero(subArea)[2]

    mean = np.array([0,0,0],dtype='int64')
    total = 0
    for i in range(len(index_r)):
        mean = mean + ( np.array([index_r[i]+idx1[0],index_g[i]+idx1[1],index_b[i]+idx1[2]]) #
                                                                    * subArea[index_r[i],index_g[i],index_b[i]] )
        total += subArea[index_r[i],index_g[i],index_b[i]]
    mean = np.round( mean / total).astype('int64')
```

(4) 새로 계산된 mean과 이전 mean사이의 거리를 계산해 거의 차이가 없는 경우 반복을 종료한다. Distance 계산에 L^1 norm을 사용해서 계산 속도를 빠르게 할 수 있도록 했다.

```
# previous, current mean 사이의 거리 비교
d = distance(mean, previous_mean)
previous_mean = mean
# 거리가 가까운 경우 종료
if d < 5:
    meanList.append(mean)
    break
```

```
def distance(a,b):  # L^1 norm
    return np.sum(np.abs(a-b))
```

(5) 반복이 끝나고 바로 centroid list를 확인하면 같은 값이 여러 개로 반환되기도하고 매우 가까운 점이 반환되기도 한다. 따라서 이런 인접한 점들을 하나로 묶어서 최종 centroid list를 반환한다.

```
# mean으로 계산된 centroid list를 반환
meanList = np.array(meanList)
for i in range(len(meanList)):
    for j in range(i+1, len(meanList)):
        d = distance(meanList[i], meanList[j])
        if d < 10:
            meanList[j] = meanList[i]

return np.unique(meanList,axis=0)
```

```
preprocessing
[[106  72  84]
 [106  72  84]
 [104  71  83]
 [106  72  84]
 [106  72  84]
 [106  72  84]
 [106  72  84]
 [106  72  84]
 [106  72  84]
 [106  72  84]
 [106  72  84]
 [106  72  84]
 [107  74  85]
 [106  72  84]
 [104  71  83]
 [104  71  83]
 [108  76  86]
 [106  73  84]
 [103  69  82]
 [108  76  87]
 [106  73  85]
 [107  75  86]
 [108  76  86]
 [165 106  97]
 [108  76  87]
 [108  75  86]
 [108  75  86]
 [108  76  87]
 [136  94  95]
 [108  76  86]
 [108  76  87]
 [137  94  95]
 [165 106  97]
 [165 106  97]
 [165 106  97]
 [165 106  97]
 [149 174 185]
 [141 144 145]
 [158 197 221]
 [139 166 162]
 [165 106  97]
 [165 106  97]
 [165 106  97]
 [165 106  97]
 [165 106  97]
 [165 106  97]
 [166 106  97]
 [166 106  97]
 [158 197 221]
 [158 197 221]
 [166 106  97]
 [166 106  97]
 [166 106  97]
 [166 106  97]
 [160   0 223]
 [158 197 221]
 [166 106  97]
 [166 106  97]
 [166 106  97]
 [140 167 163]
 [166 106  97]
 [166 106  97]
 [166 106  97]
 [166 106  97]
 [166 106  97]
 [158 197 221]
 [211 221 216]
 [212 222 217]]
```

```
result
[[106  72  84]
 [108  76  87]
 [136  94  95]
 [139 166 162]
 [141 144 145]
 [149 174 185]
 [158 197 221]
 [160   0 223]
 [165 106  97]
 [211 221 216]]
```

(6) 각 이미지 픽셀에서 가장 가까운 centroid값으로 이미지 픽셀 값을 대체한다.

```
# 각 cluster에 centroid 값 할당
def clusterring(img, centroids):
    h,w = img.shape[:2]
    result = img.copy()
    for i in range(h):
        for j in range(w):
            M = 1000
            temp = [0,0,0]
            for centroid in centroids:
                d = distance(centroid, img[i,j])
                if d < M:
                    M=d
                    temp = centroid
            result[i,j,0] = temp[0]
            result[i,j,1] = temp[1]
            result[i,j,2] = temp[2]
    return result
```

<mean-shift 결과 출력/분석>

Mean-shift의 경우 K-means와 다르게 k값을 parameter로 넣어주지 않아도 결과 출력이 가능하다. 따라서 함수의 인수로 이미지만 넣어주어도 centroid계산이 가능하다.

- centroids 출력: 총 10개의 centroid가 결과로 출력된 것을 확인할 수 있다. K-mean에 k값을 10으로 넣어 출력 값을 확인해보면 오른쪽 출력을 확인할 수 있는데 비교해보면 유사한 값을 갖는 경우도 있지만 확연히 다른 값을 갖는 경우도 존재했다.

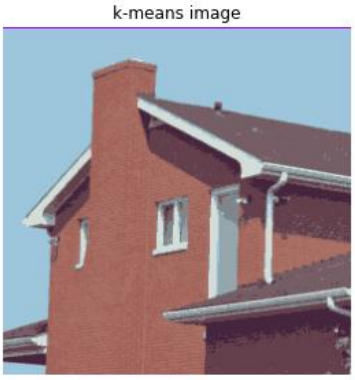
< mean-shift >

```
[[105  72  84]
 [108  76  87]
 [136  94  95]
 [139 156 162]
 [141 144 145]
 [149 174 185]
 [158 197 221]
 [160   0 223]
 [165 106  97]
 [211 221 216]]
```

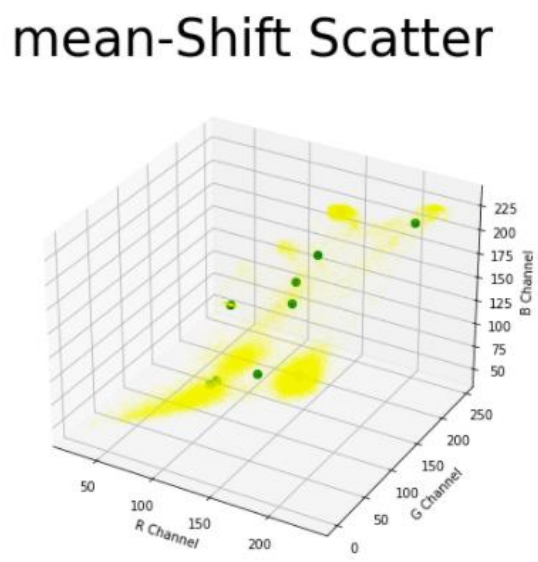
< k-mean, k=10>

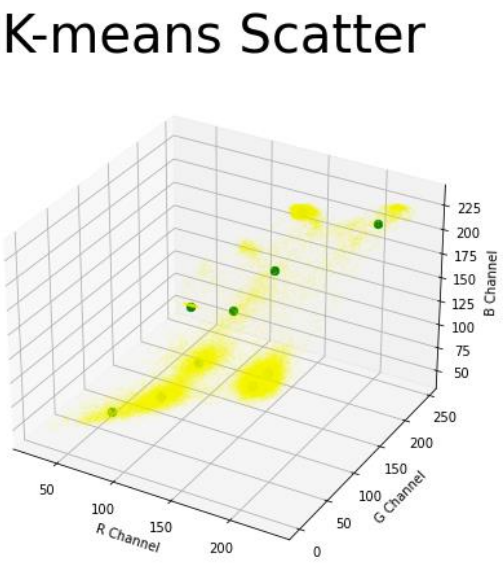
```
[[128 135 138]
 [ 90 100 162]
 [ 37  63  70]
 [149 166 172]
 [ 94 102 118]
 [212 215 221]
 [ 63  76 100]
 [157 197 220]
 [101 111 170]
 [  0 159 222]]
```

- 결과 이미지 출력: 영역 분할이 정상적으로 진행된 것을 확인할 수 있다.



- Scattering: 위에서 언급했던 대로 유사한 값을 갖는 경우도 있지만 확연히 다른 값을 갖는 경우도 존재했다.





2. (Local Descriptor) Based on the descriptor designed in FL class, design the followings.

data processing

MNIST dataset 개수는 총 70000개인데 이를 모두 사용하면 수행시간이 너무 오래 걸리기 때문에 총 dataset에 대해 10000개, 30000개의 subset data를 뽑아서 subset에 대한 k-means를 2번 수행했다.

```
# data load
mnist = dataset.fetch_openml('mnist_784')
mnist_data = mnist.data.to_numpy()
mnist_target = mnist.target.to_numpy()
print('mnist data shape : ',mnist_data.shape)
print('mnist target shape : ',mnist_target.shape)
```

```
mnist data shape :  (70000, 784)
mnist target shape :  (70000,)
```

입력된 data의 개수 중 원하는 개수의 data를 랜덤으로 뽑는 함수를 구현했다.

```
def select(data, target, number=10000):
    subset_idx = np.sort( np.random.choice(len(data), number,replace=False) )

    subset_data = data[subset_idx]
    subset_target = target[subset_idx]
    return subset_data, subset_target

subset_data, subset_target  = select(mnist_data, mnist_target, number=20000)
```

1) Apply k-means (k=10) algorithm to MNIST test dataset, where input dimension is 282=784.

```
k =10
def kmeans(data, k, feature_size = 784, feature_scale = 256, centroid=None,):
    data = data.copy()
    repeat = 1
    if centroid == None:
        centroid = np.random.uniform(0,feature_scale,(k,feature_size)) # random initi
    print('start')

    while True:
        repeat+=1

        # groups, indexes
        groups, indexes = [], []

        # cluster 개수에 맞게 리스트 추가
        for d in range(k):
            groups.append([])
            indexes.append([])

        # 거리에 따라 group에 각 feature 값과 index 추가
        for i in range(len(data)):
            distances = np.array([])
            for d in range(k):
                distances = np.append(distances, np.sum((data[i] - centroid[d])**2))
            idx=np.argmin(distances)
            groups[idx].append(data[i])
            indexes[idx].append(i)

        # 각 그룹의 feature 평균값으로 다음 centroid 계산
        next_centroid = np.zeros( (k,feature_size) )
        for d in range(k):
            if len(indexes[d])!=0:
                next_centroid[d] = np.mean(np.array(groups[d]),axis=0)
        if repeat % 20 == 0:
            print('repeat:', repeat)

        # 종료조건
        if(np.array_equal(centroid, next_centroid)):
            print('find!')
            break
        centroid = next_centroid

    print('repeat:', repeat)
    return centroid, indexes
```

```
centroid, group_indexes = kmeans(subset_data, 10, 784, 256)
```

k-mean의 경우 1번에서 사용했던 방법과 거의 동일하다.

인수를 여러가지 추가하였는데, feature의 dimension을 다르게 입력할 수 있도록 했고 cetroid initiallization이 k-means의 성능에 큰 영향을 미치므로 centroid를 랜덤 생성할 경우 feature scale을 넣어주거나 외부에서 centroid를 입력으로 넣어줄 수 있도록 함수를 만들었다.

clustering결과 10000개의 dataset에 대해 74번의 반복, 30000개의 dataset에 대해 100번의 반복을 수행, centroid를 결정했다.

```
start
repeat: 20
repeat: 40
repeat: 60
find!
repeat: 74
```

```
start
repeat: 20
repeat: 40
repeat: 60
repeat: 80
repeat: 100
find!
repeat: 100
```

```
subset_target[group_indexes[2]]
```

```
array(['0', '0', '0', '0', '0', '0', '0', '0', '0', '6', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '6', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '6', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '5', '0', '8',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '5',
       '0', '0', '0', '0', '0', '0', '5', '0', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '8',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '0', '9', '0', '0', '0', '0', '0', '0', '0', '5',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '5', '8', '0', '0', '8', '0', '0', '3', '8', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '0', '3', '0', '0', '0', '0', '0', '8', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '9', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '5', '0', '0', '0', '0', '0', '9', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '9', '0', '0', '0', '0', '0', '5', '0', '0', '0', '0',
       '0', '0', '0', '0', '6', '0', '0', '0', '0', '0', '0', '0',
       '0', '5', '0', '0', '0', '0', '2', '0', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '7', '0', '0', '0', '0', '0',
       '0', '5', '0', '0', '0', '0', '0', '9', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '6', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '8', '0', '0', '6', '0', '0',
       '5', '0', '0', '0', '0', '0', '5', '0', '0', '0', '0', '0',
       '0', '9', '0', '0', '0', '0', '3', '0', '0', '0', '6', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '6', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '9', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '5', '0', '0', '0', '0',
       '0', '0', '5', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
       '0', '8', '0', '0', '0', '0', '0', '0', '0', '0'],
      dtype=object)
```

clustering된 index를 사용, Boolean indexing을 수행해 어떤 값들이 grouping되었는지 결과를 확인해 보았다.

간간히 다른 값이 clustering된 것도 확인할 수 있었지만 가장 빈번한 값으로 0이 나타나는 것으로 보아 이 cluster의 대푯값은 '0' 임을 알 수 있다.

각 cluster가 어떤 값을 나타내는 cluster인지 알아내야 하는데 각 cluster 내부의 값들로 histogram을 만들고 가장 많이 나타나는 값을 각 cluster의 대푯값으로 지정했다.

다음은 클러스터의 대푯값을 나타낸 결과이다. 10000개, 30000개 모두 5는 정상적으로 clustering 되지 않았음을 예상할 수 있다.

[6 0 7 6 1 4 3 8 1 2] [1 7 3 6 8 4 2 1 0 0]

```
cluster_target = []
cluster_target.append(np.argmax(np.histogram((subset_target[group_indexes[0]]).astype(int),bins=[0, 1, 2, 3,4,5,6,7,8,9,10])[0]) )
cluster_target.append(np.argmax(np.histogram((subset_target[group_indexes[1]]).astype(int),bins=[0, 1, 2, 3,4,5,6,7,8,9,10])[0]) )
cluster_target.append(np.argmax(np.histogram((subset_target[group_indexes[2]]).astype(int),bins=[0, 1, 2, 3,4,5,6,7,8,9,10])[0]) )
cluster_target.append(np.argmax(np.histogram((subset_target[group_indexes[3]]).astype(int),bins=[0, 1, 2, 3,4,5,6,7,8,9,10])[0]) )
cluster_target.append(np.argmax(np.histogram((subset_target[group_indexes[4]]).astype(int),bins=[0, 1, 2, 3,4,5,6,7,8,9,10])[0]) )
cluster_target.append(np.argmax(np.histogram((subset_target[group_indexes[5]]).astype(int),bins=[0, 1, 2, 3,4,5,6,7,8,9,10])[0]) )
cluster_target.append(np.argmax(np.histogram((subset_target[group_indexes[6]]).astype(int),bins=[0, 1, 2, 3,4,5,6,7,8,9,10])[0]) )
cluster_target.append(np.argmax(np.histogram((subset_target[group_indexes[7]]).astype(int),bins=[0, 1, 2, 3,4,5,6,7,8,9,10])[0]) )
cluster_target.append(np.argmax(np.histogram((subset_target[group_indexes[8]]).astype(int),bins=[0, 1, 2, 3,4,5,6,7,8,9,10])[0]) )
cluster_target.append(np.argmax(np.histogram((subset_target[group_indexes[9]]).astype(int),bins=[0, 1, 2, 3,4,5,6,7,8,9,10])[0]) )
```

2) Measure the clustering accuracy by counting the number of incorrectly grouped images, and analyze the results.

각 cluster의 대푯값과 cluster 내부의 값들을 비교해서 같은 값을 가지는 개수를 누적하고 전체 개수로 나눠 정확도를 파악한다. 10000개의 data로 수행한 결과 약 60%의 정확도로 clustering된 것을 확인할 수 있었고 30000개의 data로 수행한 결과 약 58%의 정확도로 clustering되었다.

```
for i in range(10):
    count += len(np.where(subset_target[group_indexes[i]] == str(cluster_target[i]) )[0])
print('accuracy:',100 * (count / len(subset_data)), '%')
```

```
accuracy: 60.36 %
```

```
accuracy: 58.57 %
```

% k-means 함수에 initial centroid옵션을 만들어 두었기 때문에 initial centroid를 각 0~9까지의 값을 나타내는 784크기의 dataset의 input으로 초기화해서 k-means를 수행했다. 향상된 결과를 기대했지만 별다른 차이가 없었다. 다음은 initial centroid를 생성하는 코드로 target값을 통해 input을 찾아낸 후 초기화했다.

```
initial_centroid = []
initial_centroid.append(subset_data[np.where(subset_target=='0')[0][3]])
initial_centroid.append(subset_data[np.where(subset_target=='1')[0][3]])
initial_centroid.append(subset_data[np.where(subset_target=='2')[0][3]])
initial_centroid.append(subset_data[np.where(subset_target=='3')[0][3]])
initial_centroid.append(subset_data[np.where(subset_target=='4')[0][3]])
initial_centroid.append(subset_data[np.where(subset_target=='5')[0][3]])
initial_centroid.append(subset_data[np.where(subset_target=='6')[0][3]])
initial_centroid.append(subset_data[np.where(subset_target=='7')[0][3]])
initial_centroid.append(subset_data[np.where(subset_target=='8')[0][3]])
initial_centroid.append(subset_data[np.where(subset_target=='9')[0][3]])
initial_centroid = np.array(centroid)
print(initial_centroid.shape)
```

```
(10, 784)
```

3) Divide the input 28×28 image into four 14×14 sub-blocks, and compute the histogram of orientations for each sub-block as below.

```
def devide(image):
    subMatrix = []
    subMatrix.append(image[0:14,14:28])
    subMatrix.append(image[14:28,0:14])
    subMatrix.append(image[0:14,0:14])
    subMatrix.append(image[14:28,14:28])
    subMatrix = np.array(subMatrix)
    return subMatrix
```

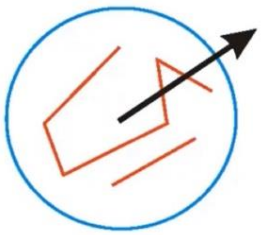
다음은 28x28 image를 입력으로 받아 4개의 subMatrix로 나누어 출력하는 함수이다.

각각의 SubMatirx에 대해 Gaussian Filtering을 취한 뒤 magnitude와 direction Matrix를 생성하고 histogram을 만드는 과정을 진행한다.

먼저 Gaussian Filtering을 취한다. Gradient를 구하는 과정에서 noise에 의해 값이 크게 튀는 것을 막아준다.

```
def orientation_histogram(img):
    #Gaussian filtering
    Gaussian_kernel = gaussianFilter(3,3)
    img = filtering(img, Gaussian_kernel)
```

강의자료의 SIFT기술자에서 사용하는 수식을 그대로 사용해서 magnitude, direction Matrix를 완성한다.



$$L(x,y,k\sigma) = G(x,y,k\sigma) * I(x,y)$$

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \text{atan2}(L(x,y+1) - L(x,y-1), L(x+1,y) - L(x-1,y))$$

```
# magnitude, direction Matrix 생성
h, w = img.shape
magnitude = np.zeros((h,w))
direction = np.zeros((h,w))
img = np.pad(img, ((1,1),(1,1)), 'symmetric')
for i in range(h):  # y
    for j in range(w): #x
        temp_y = i+1
        temp_x = j+1
        magnitude[i,j] = np.sqrt((img[ temp_y, temp_x+1] - img[temp_y, temp_x-1])**2 + \
                                 (img[temp_y+1,temp_x] - img[temp_y-1,temp_x])**2)
        direction[i,j] = np.rad2deg(np.arctan2(img[temp_y+1,temp_x] - img[temp_y-1,temp_x], \
                                 img[temp_y, temp_x+1] - img[temp_y, temp_x-1] ))
```

다음은 히스토그램을 생성하는 코드이다. 가독성을 위해 degree로 변환해서 코딩을 진행했고 2차원 배열을 1차원으로 펼쳐서 반복문을 통해 히스토그램을 생성했다. -180 ~ 180 값을 0~360 범위로 변환했는데 이는 histogram index로 사용할 경우 음수가 나오면 안되기 때문이다.

```
# histogram 생성
bins =8
width = np.rad2deg((np.pi*2 ) / bins)          # 45도
histogram = np.zeros(bins)
temp_mag = magnitude.ravel()                   # 반복문 사용 편리를 위해 펼침
direction = direction.ravel()
direction[ direction < 0 ] += 360              # -180 ~ 180 -> 0 ~ 360으로 변환
hist_index = (direction // width).astype(int)  # 각 direction값을 45로 나눈 몫을 histogram index로 사용
hist_index[hist_index==8] = 0                  # 360도 == 0도
for i in range( len(hist_index) ):
    histogram[hist_index[i]] += temp_mag[i]    # magnitude 누적
```

```
# feature Extraction
subset_feature = []
for i in range(len(subset_data)):
    temp = []
    subMatrix = devide(subset_data[i].reshape(28,28))
    for j in range(len(subMatrix)):
        temp.append(orientation_histogram(subMatrix[j]))
    temp = np.array(temp).ravel()
    subset_feature.append(temp)
subset_feature  = np.array(subset_feature)
```

subset데이터를 4개의 submatrix로 나누고 각 submatrix의 histogram을 생성해 이어 붙여 feature를 만들어내는 과정을 반복한다.

그 결과 모든 data에 대해 32 dimension의 feature가 추출된 것을 확인할 수 있다.

```
subset_feature shape:  (10000, 32)  subset_feature shape:  (30000, 32)
```

4) Apply k-means (k=10) algorithm again using feature, where input dimension is 8×4=32.

10000개에 대해 33번의 반복, 30000개에 대해서는 42번의 반복을 통해 centroid를 결정했다. 앞서 784 차원의 input에 대해 구현했던 k-means를 인수(feature size, feature scale)을 바꿔 넣어 그대로 실행해 k-means를 수행했다. (1)번에서 784차원의 input에 대해 수행했던 k-means의 경우 총 74번/100번의 반복을 수행했는데 반복횟수 및 수행시간이 단축된 것을 확인할 수 있다.

```
centriod, group_indexes = kmeans(subset_feature, 10, feature_size = 32, feature_scale = 5000)
```

```
start
repeat: 20
find!
repeat: 33
```

```
start
repeat: 20
repeat: 40
find!
repeat: 42
```

5) Measure the clustering accuracy for feature-based approach, and analyze the results.

마찬가지로 각 cluster의 대푯값을 찾았고 확인해본 결과 10000개 데이터에 대한 결과는 9를 clustering하지 못했고 30000개에 대해서는 0~9까지 모든 수가 대푯값으로 나타난 것을 확인할 수 있었다. 하지만 정확도의 경우 784개의 feature를 input 사용한 결과와 거의 차이가 없었다.

<1000개>

```
[4, 7, 3, 2, 1, 1, 0, 8, 6, 5]
accuracy: 59.260000000000005 %
```

<30000개>

```
[0, 6, 3, 8, 9, 5, 7, 1, 4, 2]
accuracy: 57.43666666666667 %
```

두 가지 케이스의 가장 큰 차이는 featured의 dimension으로, 784 x N, 32 x N로 생각할 수 있는데 feature extraction의 경우 약 1/25배의 크기 dimension 가지고도 거의 비슷한 clustering 성능을 보일 수 있다는 점에서 feature extraction이 잘 되었다고 생각할 수 있다. 다른 말로 표현하면 784차원의 data가 32차원으로 잘 압축되었다고도 말할 수 있다.

3. (Back propagation) Design the simple single-layer perceptron (SLP) network for IRIS dataset.

1) Use python library to extract images, and separate them into test and training set.

iris dataset을 **sklearn library**를 통해 load한다.

```
iris = dataset.load_iris()

iris_data = iris.data
iris_target = iris.target
```

training set과 test set으로 분리하는데 전체 데이터 셋에 대한 테스트 셋의 비율을 parameter로 준다. 계산한 test set의 개수만큼 index를 랜덤으로 뽑아낸 뒤에 전체 집합 index와 뽑아낸 index의 차집합을 통해서 training set의 index를 구한다. 그 이후 data와 target에 대해 Boolean indexing을 수행하면 성공적으로 test set과 training set을 분리할 수 있다. 가독성을 위해 sort를 수행한 뒤 shape와 index를 출력했는데 겹치는 index 없이 정상적으로 분리된 것을 확인할 수 있다.

```
def separate(data,target,testset_rate=0.2):
    rate = testset_rate

    testset_idx = np.sort( np.random.choice(len(data), round( len(data)* rate),replace=False) )
    trainingset_idx = np.sort( np.setdiff1d(np.arange(len(data)), testset_idx) )

    print('testset_idx \n')
    print(testset_idx)
    print('\n trainingset_idx \n')
    print(trainingset_idx)

    test_data = data[testset_idx]
    test_target = target[testset_idx]

    training_data = data[trainingset_idx]
    training_target = target[trainingset_idx]

    print('\n test set shape:  ', test_data.shape, test_target.shape )
    print('\n training set shape:  ', training_data.shape, training_target.shape )
    return test_data, test_target, training_data,training_target
```

```
iris data shape :  (150, 4)
iris target shape :  (150,)

testset_idx

[  0   1   2   3   5  10  14  20  34  44  47  51  52  56  60  61  64  81
  85  90  91 102 103 106 109 115 116 120 140 147]

 trainingset_idx

[  4   6   7   8   9  11  12  13  15  16  17  18  19  21  22  23  24  25
  26  27  28  29  30  31  32  33  35  36  37  38  39  40  41  42  43  45
  46  48  49  50  53  54  55  57  58  59  62  63  65  66  67  68  69  70
  71  72  73  74  75  76  77  78  79  80  82  83  84  86  87  88  89  92
  93  94  95  96  97  98  99 100 101 104 105 107 108 110 111 112 113 114
 117 118 119 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135
 136 137 138 139 141 142 143 144 145 146 148 149]

 test set shape:   (30, 4) (30,)

 training set shape:   (120, 4) (120,)
```

2) Select any two datasets for training and test.

np.random.choice를 사용해서 각 집합에 대한 index를 랜덤으로 선택하고 index를 사용해 training set과 test set에서 data와 target을 하나씩 뽑아내 결과를 확인한다.

```
random_index1 = np.random.choice(len(training_data),1)
training_sample = training_data[random_index1]
training_target = training_target[random_index1]
random_index2 = np.random.choice(len(test_data),1)
test_sample = test_data[random_index2]
test_target = test_target[random_index2]
print(training_sample, training_target)
print(test_sample, test_target)
```

```
[[6.1 3.  4.9 1.8]] [2]
[[5.5 2.6 4.4 1.2]] [1]
```

3) Design SLP network to classify them. Use the sigmoid function for activation function.

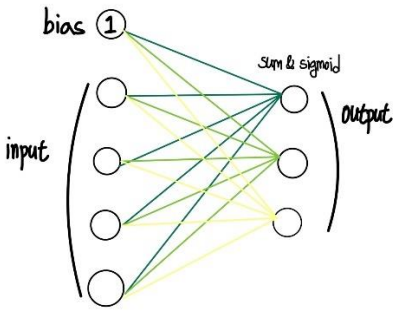
Sigmoid 함수의 수식대로 activation function을 구현했다.

```
# sigmoid
def sigmoid(x):
    return 1 / (1 + np.exp(-x))     # sigmoid
```

$$s(z) = \frac{1}{1+e^{-z}}$$

SLP를 객체로 생성할 수 있게 했다. iris dataset의 feature길이는 4이고 bias를 하나 추가한다. target의 class 개수는 3개 이므로 input size는 5, output size는 3, 5X3 크기의 weight를 갖는 NN 생성한다. Forward propagation의 동작은 input과 weight의 weighted sum을 수행하고 activation function을 통과시켜 출력하는데 이를 행렬 곱을 사용해서 간단하게 구현할 수 있다.

```
class NN():
    def __init__(self, input_size, output_size):
        self.input_size = input_size
        self.output_size = output_size
        # weights
        self.weight = np.random.randn(input_size, output_size)
        print('input size:    ', self.input_size)
        print('weight shape: ', self.weight.shape)
        print('output size:   ', self.output_size)

    # Forward propagation
    def forward(self, x):
        self.output_in  = np.dot(x, self.weight)
        self.output_out = sigmoid(self.output_in)
        return self.output_out
```




```
input_size, output_size= 5, 3
NN = NN(input_size, output_size)
```

<NN 선언>

(2)번에서 뽑아낸 iris data를 입력으로 넣기 이전에 bias를 1로 추가하여 크기 5를 갖는 input으로 만든다. 추후에 error를 계산하기 위해서 one-hot 인코딩을 사용하기 때문에 미리 구현을 해서 predict값과 target을 비교했다. 출력이 벡터로 나오는 경우 sigmoid의 출력이 곧 확률이기 때문에 가장 큰 값을 최종 predict값으로 선택한다. 아래 결과를 보면 training sample의 경우 정상적으로 예측했지만 test sample의 경우 정상적으로 예측하지 못한 것을 확인할 수 있다.

```python
# 학습 이전
def one_hot_encoding(input_, num_of_class):
    encoding = np.zeros(num_of_class)
    encoding[input_]=1
    return encoding

input_ = np.append(training_sample, 1)
print('input:   ', input_)
target= one_hot_encoding(training_target, 3)
print('target:   ', target)
predict = NN.forward(input_)
print('predict: ', np.round_(predict, 4))
print()

input_ = np.append(test_sample, 1)
print('input:   ', input_)
target= one_hot_encoding(test_target, 3)
print('target:   ', target)
predict = NN.forward(input_)
print('predict: ', np.round_(predict, 4))
```

```
input:    [6.1 3.  4.9 1.8 1. ]
target:    [0. 0. 1.]
predict:  [0.     0.0023 0.9566]

input:    [5.5 2.6 4.4 1.2 1. ]
target:    [0. 1. 0.]
predict:  [0.     0.0192 0.2883]
```

4) Use the back propagation method to train the network parameters.

Back-propagation의 경우 다음과 같이 계산할 수 있는데 왼쪽부터 순서대로 Error function, activation function, weighted sum에 대한 미분을 의미한다. Sum의 weight에 대한 미분은 곧 weight에 곱해지는 입력을 의미하며 MSE에 대한 미분의 경우 상수가 2/n이 상수로 붙어야 하지만 Learning rate를 곱하면 상수는 큰 의미를 갖지 못하므로 predict - target으로 간단하게 구현했다. Sigmoid의 미분은 아래 수식과 같으며 쉽게 구현 가능하다.

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial x_i}\frac{\partial x_i}{\partial s_i}\frac{\partial s_i}{\partial w_{ji}}$$

```python
# sigmoid 미분
def differ_sigmoid(x):
    return sigmoid(x)*(1-sigmoid(x))

# MSE 미분
def differ_MSE(predict, target):
    return predict - target
```

$$\frac{d}{dx}sigmoid(x) = sigmoid(x)(1 - sigmoid(x))$$

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

다음은 Back-propagation의 구현이다. 차례로 MSE(loss function)의 미분, sigmoid(activation function)의 미분을 곱하고 input과 행렬 곱을 취하면 weight의 shape과 같은 gradient를 구할 수 있고 learning rate를 곱해서 weight에서 빼면(gradient의 반대 방향 이동) backpropagation을 구현할 수 있다. 이를 통해 loss function의 minimum을 찾을 수 있고 모델을 학습시킬 수 있다.

```python
def backward(self, input_, label, predict):
    error_diff = differ_MSE(predict, label)                  # loss function differectial
    sig_diff = error_diff * differ_sigmoid(predict)          # sigmoid differectial
    gradient = np.expand_dims(input_, axis=0).T @ np.expand_dims(sig_diff, axis=0)          # sum differential
    self.weight -= gradient * 0.1                            # learning rate = 0.1
```

5) Compute the training and test error for every epoch

학습 코드를 다음과 같이 작성했고 주석으로 각 과정의 설명을 달아 놓았다. 또한 print문을 적극적으로 사용해서 학습 로그를 확인할 수 있게 했다. training 코드와 test 코드의 가장 큰 차이점으로 back-propagation 과정의 존재 유무이며 test코드에서는 삭제했다. Loss 계산은 MSE loss를 사용했다.

```
epoch = 500
for j in range(epoch):
    print('='*80)
    print('epoch: ', j+1)
    number = 0
    average_loss = 0
    for i in range(len(training_data)):                    # training set
        input_ = training_data[i]
        input_ = np.append(input_, 1)                      # input에 bias 추가
        label1_ = training_target[i]
        label1_ = one_hot_encoding(label1_,3)              # target one-hot encoding
        predict = NN.forward(input_)                       # predict
        correct = np.argmax(predict)==np.argmax(label1_)   # 정답 여부 확인
        if correct:
            number+=1
        if i%20 == 0:                                      # for visualization
            print(i,'번째 predict:', np.round_(predict, 3),'    label:', label1_, correct)
        average_loss += MSEloss(label1_, predict)          # loss 누적
        NN.backward(input_, label1_, predict)              # back-propagation, 학습
    print()
    print('='*80)
    print('< training set >')                              # for visualization
    print('average loss: ', 100*(average_loss/len(training_data)))
    print('accuracy: ', 100*(number/len(training_data)), '%')

    number = 0
    for i in range(len(test_data)):                        # test set
        input_ = test_data[i]
        input_ = np.append(input_, 1)
        label1_ = test_target[i]
        label1_ = one_hot_encoding(label1_,3)
        predict = NN.forward(input_)
        correct = np.argmax(predict)==np.argmax(label1_)
        if correct:
            number+=1
        average_loss += MSEloss(label1_, predict)          # back-propagation 삭제됨
    print()
    print('='*80)
    print('< test set >')
    print('average loss: ', 100*(average_loss/len(test_data)))
    print('accuracy: ', 100*(number/len(test_data)), '%')
```

```
# MSE(loss function)
def MSEloss(label, predict):
    return np.mean(np.square(label - predict))
```

<학습 로그>

epoch가 증가함에 따라 training set과 test set에 대한 Accuracy가 점점 증가하는 것을 확인할 수 있었다. 학습이 성공적으로 수행되고 있음을 예상할 수 있다. 모델의 성능은 Weight initialization이 어떻게 수행됐는지에 따라 영향이 있다. 이는 Gradient decent를 수행해 cost function의 극소점을 찾는 과정에서 극소점이 여러 개 존재할 수 있기 때문이다. 이를 감안해 model를 다시 초기화해서 학습을 다시 수행했는데 예상한 대로 학습결과가 다르게 나온 것을 확인할 수 있었다.

<SLP에 대한 수행>

첫번째 시행

```
================================================================================
epoch:  1
0 번째 predict: [0.078 0.226 0.398]     label: [1. 0. 0.] False
20 번째 predict: [0.94  0.067 0.08 ]     label: [1. 0. 0.] True
40 번째 predict: [0.994 0.561 0.001]     label: [0. 1. 0.] False
60 번째 predict: [0.027 0.948 0.007]     label: [0. 1. 0.] True
80 번째 predict: [0.01  0.837 0.015]     label: [0. 0. 1.] False
100 번째 predict: [0.01  0.02  0.959]     label: [0. 0. 1.] True

================================================================================
< training set >
average loss:  5.255866978956892
accuracy:  91.66666666666666 %

================================================================================
< test set >
average loss:  68.6669142527366
accuracy:  26.666666666666668 %
================================================================================
================================================================================
epoch:  180
0 번째 predict: [0.994 0.256 0.   ]     label: [1. 0. 0.] True
20 번째 predict: [0.989 0.103 0.   ]     label: [1. 0. 0.] True
40 번째 predict: [0.     0.163 0.135]     label: [0. 1. 0.] True
60 번째 predict: [0.009 0.989 0.   ]     label: [0. 1. 0.] True
80 번째 predict: [0.     0.85  0.703]     label: [0. 0. 1.] False
100 번째 predict: [0.     0.008 0.81 ]     label: [0. 0. 1.] True

================================================================================
< training set >
average loss:  2.3617088392445313
accuracy:  95.83333333333334 %

================================================================================
< test set >
average loss:  21.272335853392477
accuracy:  73.33333333333333 %
================================================================================
```

```
================================================================================
epoch:  30
0 번째 predict: [0.951 0.164 0.001]     label: [1. 0. 0.] True
20 번째 predict: [0.948 0.062 0.001]     label: [1. 0. 0.] True
40 번째 predict: [0.003 0.135 0.447]     label: [0. 1. 0.] False
60 번째 predict: [0.038 0.969 0.005]     label: [0. 1. 0.] True
80 번째 predict: [0.     0.749 0.717]     label: [0. 0. 1.] False
100 번째 predict: [0.001 0.02  0.889]     label: [0. 0. 1.] True

================================================================================
< training set >
average loss:  2.4238187233613098
accuracy:  95.0 %

================================================================================
< test set >
average loss:  27.87241171412121
accuracy:  66.66666666666666 %
================================================================================
================================================================================
epoch:  500
0 번째 predict: [0.998 0.283 0.   ]     label: [1. 0. 0.] True
20 번째 predict: [0.995 0.116 0.   ]     label: [1. 0. 0.] True
40 번째 predict: [0.     0.162 0.026]     label: [0. 1. 0.] True
60 번째 predict: [0.004 0.992 0.   ]     label: [0. 1. 0.] True
80 번째 predict: [0.     0.89  0.914]     label: [0. 0. 1.] True
100 번째 predict: [0.     0.005 0.764]     label: [0. 0. 1.] True

================================================================================
< training set >
average loss:  2.596805128442283 3
accuracy:  99.16666666666667 %

================================================================================
< test set >
average loss:  21.16815546760285
accuracy:  80.0 %
```

두번째 시행

두번째 시행에서 학습과정의 진척도가 다르게 나타날 뿐만 아니라 최종 500번째 epoch에서는 test set에 대한 정확도가 76.67%으로 첫번째 시행에 비해 최종 결과가 약 4% 낮게 나왔다.

```
================================================================================
epoch:  1
0 번째 predict: [0.999 0.414 0.073]     label: [1. 0. 0.] True
20 번째 predict: [0.999 0.033 0.031]     label: [1. 0. 0.] True
40 번째 predict: [0.999 0.022 0.012]     label: [1. 0. 0.] True
60 번째 predict: [0.034 0.966 0.003]     label: [0. 1. 0.] True
80 번째 predict: [0.003 0.996 0.002]     label: [0. 0. 1.] False
100 번째 predict: [0.002 0.018 0.964]     label: [0. 0. 1.] True

================================================================================
< training set >
average loss:  4.723873044786875
accuracy:  91.66666666666666 %

================================================================================
< test set >
average loss:  57.229304329164584
accuracy:  33.33333333333333 %
================================================================================
================================================================================
epoch:  180
0 번째 predict: [0.997 0.199 0.   ]     label: [1. 0. 0.] True
20 번째 predict: [0.993 0.081 0.   ]     label: [1. 0. 0.] True
40 번째 predict: [0.999 0.012 0.   ]     label: [1. 0. 0.] True
60 번째 predict: [0.     0.976 0.046]     label: [0. 1. 0.] True
80 번째 predict: [0.     0.85  0.996]     label: [0. 0. 1.] True
100 번째 predict: [0.     0.023 0.926]     label: [0. 0. 1.] True

================================================================================
< training set >
average loss:  2.1988249957321506
accuracy:  97.5 %

================================================================================
< test set >
average loss:  24.192147045931364
accuracy:  70.0 %
================================================================================
```

```
================================================================================
epoch:  30
0 번째 predict: [0.975 0.105 0.001]     label: [1. 0. 0.] True
20 번째 predict: [0.968 0.036 0.001]     label: [1. 0. 0.] True
40 번째 predict: [0.994 0.017 0.   ]     label: [1. 0. 0.] True
60 번째 predict: [0.001 0.962 0.049]     label: [0. 1. 0.] True
80 번째 predict: [0.     0.985 0.559]     label: [0. 0. 1.] False
100 번째 predict: [0.     0.024 0.941]     label: [0. 0. 1.] True

================================================================================
< training set >
average loss:  2.338802704802489
accuracy:  95.0 %

================================================================================
< test set >
average loss:  29.94665365957001
accuracy:  60.0 %
================================================================================
================================================================================
epoch:  500
0 번째 predict: [0.999 0.237 0.   ]     label: [1. 0. 0.] True
20 번째 predict: [0.997 0.099 0.   ]     label: [1. 0. 0.] True
40 번째 predict: [1.    0.01 0.   ]     label: [1. 0. 0.] True
60 번째 predict: [0.     0.983 0.04 ]     label: [0. 1. 0.] True
80 번째 predict: [0.     0.768 1.   ]     label: [0. 0. 1.] True
100 번째 predict: [0.     0.019 0.925]     label: [0. 0. 1.] True

================================================================================
< training set >
average loss:  2.3431210969524012
accuracy:  97.5 %

================================================================================
< test set >
average loss:  23.55349400743945
accuracy:  76.66666666666667 %
```

<MLP에 대한 수행>

처음 Project를 수행할 때 문제 이해에 착오가 있어 MLP(5 – 7 – 3 구조)를 먼저 구현했었다. 그 결과만 확인해보았고 다음과 같다. Layer를 추가하자 training set, test set모두에 대한 Accuracy가 SLP에 비해 높게 측정된 것을 확인할 수 있었다.

 # 첫번째 시행

```
epoch:  1
0 번째 predict: [0.62  0.375 0.155]     label: [1. 0. 0.] True
20 번째 predict: [0.726 0.271 0.126]     label: [1. 0. 0.] True
40 번째 predict: [0.865 0.159 0.083]     label: [0. 1. 0.] False
60 번째 predict: [0.592 0.408 0.079]     label: [0. 1. 0.] False
80 번째 predict: [0.38  0.633 0.084]     label: [0. 1. 0.] True
100 번째 predict: [0.255 0.429 0.242]     label: [0. 0. 1.] False

< training set >
average loss:  20.165341539337316
accuracy:  51.66666666666667 %

< test set >
average loss:  104.22258666231177
accuracy:  36.666666666666664 %

epoch:  84
0 번째 predict: [0.976 0.059 0.   ]     label: [1. 0. 0.] True
20 번째 predict: [0.97  0.071 0.   ]     label: [1. 0. 0.] True
40 번째 predict: [0.031 0.471 0.025]     label: [0. 1. 0.] True
60 번째 predict: [0.035 0.944 0.006]     label: [0. 1. 0.] True
80 번째 predict: [0.135 0.904 0.002]     label: [0. 1. 0.] True
100 번째 predict: [0.     0.185 0.867]     label: [0. 0. 1.] True

< training set >
average loss:  2.433109971816171
accuracy:  96.66666666666667 %

< test set >
average loss:  13.459921795219062
accuracy:  93.33333333333333 %
```

```
epoch:  30
0 번째 predict: [0.751 0.267 0.011]     label: [1. 0. 0.] True
20 번째 predict: [0.742 0.256 0.013]     label: [1. 0. 0.] True
40 번째 predict: [0.183 0.338 0.188]     label: [0. 1. 0.] True
60 번째 predict: [0.254 0.667 0.084]     label: [0. 1. 0.] True
80 번째 predict: [0.275 0.762 0.041]     label: [0. 1. 0.] True
100 번째 predict: [0.007 0.27  0.762]     label: [0. 0. 1.] True

< training set >
average loss:  6.3513038924515905
accuracy:  90.0 %

< test set >
average loss:  33.450370300856505
accuracy:  80.0 %

epoch:  176
0 번째 predict: [0.994 0.016 0.   ]     label: [1. 0. 0.] True
20 번째 predict: [0.991 0.024 0.   ]     label: [1. 0. 0.] True
40 번째 predict: [0.007 0.822 0.044]     label: [0. 1. 0.] True
60 번째 predict: [0.005 0.959 0.019]     label: [0. 1. 0.] True
80 번째 predict: [0.034 0.855 0.004]     label: [0. 1. 0.] True
100 번째 predict: [0.     0.115 0.897]     label: [0. 0. 1.] True

< training set >
average loss:  1.531938311807831
accuracy:  96.66666666666667 %

< test set >
average loss:  7.659781888542515
accuracy:  100.0 %
```

 # 두번째 시행

두번째 시행에서 학습과정의 진척도가 다르게 나타날 뿐만 아니라 최종 500번째 epoch에서는 test set에 대한 정확도가 93.3%으로 첫번째 시행에 비해 7% 낮게 나왔다.

```
epoch:  1
0 번째 predict: [0.74  0.773 0.702]     label: [1. 0. 0.] False
20 번째 predict: [0.813 0.495 0.38 ]     label: [1. 0. 0.] True
40 번째 predict: [0.72  0.455 0.295]     label: [0. 1. 0.] False
60 번째 predict: [0.459 0.666 0.236]     label: [0. 1. 0.] True
80 번째 predict: [0.182 0.748 0.31 ]     label: [0. 0. 1.] False
100 번째 predict: [0.161 0.537 0.503]     label: [0. 0. 1.] False

< training set >
average loss:  18.20255619947322
accuracy:  67.5 %

< test set >
average loss:  93.70038937740834
accuracy:  50.0 %

epoch:  84
0 번째 predict: [0.98  0.051 0.001]     label: [1. 0. 0.] True
20 번째 predict: [0.918 0.137 0.002]     label: [1. 0. 0.] True
40 번째 predict: [0.007 0.192 0.631]     label: [0. 1. 0.] False
60 번째 predict: [0.02  0.809 0.061]     label: [0. 1. 0.] True
80 번째 predict: [0.003 0.371 0.678]     label: [0. 0. 1.] True
100 번째 predict: [0.002 0.215 0.811]     label: [0. 0. 1.] True

< training set >
average loss:  3.6327508429843216
accuracy:  94.16666666666667 %

< test set >
average loss:  29.2064903762691
accuracy:  63.33333333333333 %
```

```
epoch:  30
0 번째 predict: [0.922 0.107 0.012]     label: [1. 0. 0.] True
20 번째 predict: [0.84  0.115 0.023]     label: [1. 0. 0.] True
40 번째 predict: [0.034 0.209 0.575]     label: [0. 1. 0.] False
60 번째 predict: [0.071 0.397 0.223]     label: [0. 1. 0.] True
80 번째 predict: [0.013 0.536 0.541]     label: [0. 0. 1.] True
100 번째 predict: [0.012 0.364 0.692]     label: [0. 0. 1.] True

< training set >
average loss:  8.772338308579975
accuracy:  84.16666666666667 %

< test set >
average loss:  52.433191096197106
accuracy:  50.0 %

epoch:  500
0 번째 predict: [0.995 0.006 0.   ]     label: [1. 0. 0.] True
20 번째 predict: [0.995 0.007 0.   ]     label: [1. 0. 0.] True
40 번째 predict: [0.001 0.973 0.003]     label: [0. 1. 0.] True
60 번째 predict: [0.001 0.995 0.   ]     label: [0. 1. 0.] True
80 번째 predict: [0.     0.007 0.991]     label: [0. 0. 1.] True
100 번째 predict: [0.     0.132 0.889]     label: [0. 0. 1.] True

< training set >
average loss:  0.9077264173032645
accuracy:  98.33333333333333 %

< test set >
average loss:  6.472436456094617
accuracy:  93.33333333333333 %
```