

# Clustering

Pattern Recognition Homeworks

Student: thu-zxs

## Solutions

### Problem 1

#### Reduction to absurdity:

Assuming there exists a  $D_k = \phi$  in the optimal solutions, with the square error  $J_e^{(1)} = J$  being minimized.

But by moving a sample, let's call  $\hat{y}$ , from a non-empty  $D_i$  to  $D_k$ , the square error corresponding to subset  $D_i$  will reduce by:

$$\frac{N_i}{N_i - 1} \|\hat{y} - m_i\|^2$$

but the square error corresponding to subset  $D_k$  will not change. ( $\|\hat{y} - m_k\| = \|\hat{y} - \hat{y}\| = 0$ )

And the total square error will be:

$$J_e^{(2)} = J - \frac{N_i}{N_i - 1} \|\hat{y} - m_i\|^2 < J = J_e^{(1)}$$

Resulting in a paradox that  $J_e^{(1)}$  is the optimal error.

So each  $D_i$  will be non-empty in optimal solution.

## Programming

### kmeans

```
def kmeans(data, label, cls_num=10):  
  
    data_size = data.shape[0]  
    data_cls = -np.ones((data_size, 1))  
  
    # Random choose samples  
    init_center_idx = np.random.choice(np.arange(data_size), size=cls_num)  
    center = data[init_center_idx]  
  
    data_cls[init_center_idx, :] = np.arange(cls_num)[:, np.newaxis]
```

```

epsilon = 1e-3
iter_cnt = 0
XSquare = np.sum(data**2, axis=1)[:, np.newaxis]
while True:
    # Computation of distance
    CSquare = np.sum(center**2, axis=1)[np.newaxis, :]
    XCCross = data.dot(center.T)
    Dist = np.sqrt(XSquare + CSquare - 2*XCCross)
    data_cls = np.argmin(Dist, axis=1)
    center_old = center.copy()
    J_e = 0
    for i in xrange(cls_num):
        clustering = data[np.where(data_cls==i)]
        if clustering.shape[0] == 0: continue
        center[i, :] = np.mean(clustering, axis=0)
        J_e += np.sum(np.sum(clustering**2, axis=1) +
np.sum(center[i,:]**2) - 2*clustering.dot(center[i,:][:, np.newaxis]))

    iter_cnt += 1
    if np.sum(np.abs(center-center_old)) < 1e-3:
        break

    print("[{}] J_e = {}".format(iter_cnt, J_e))
    print("[{}] NMI = {}".format(iter_cnt, NMI(data_cls, label)))
print(iter_cnt)
return data_cls

```

## hierarchical clustering

```

def hierarhical(data, cls_num=10):

    data_size = data.shape[0]
    data_cls = np.arange(data_size)
    data_cls_set = list(set(data_cls.tolist()))
    min_dist_clses = None
    while len(data_cls_set) > cls_num:
        min_dist = np.inf
        for i, c1 in enumerate(data_cls_set):
            cluster1 = data[np.where(data_cls==c1)]
            for c2 in data_cls_set[i+1:]:
                cluster2 = data[np.where(data_cls==c2)]
                dist = cluster_dist(cluster1, cluster2, kind="min")
                if dist < min_dist:
                    min_dist = dist
                    min_dist_clses = (c1, c2)
        data_cls[np.where(data_cls==min_dist_clses[1])] = min_dist_clses[0]
        data_cls_set = list(set(data_cls.tolist()))
    return data_cls

```

## spectral clustering

```
def spectral(data, cls_num=10, kind="cosine"):

    Cosine = data.dot(data.T)
    Cosine /= np.linalg.norm(data, axis=1)[np.newaxis, :]
    Cosine /= np.linalg.norm(data, axis=1)[:, np.newaxis]

    labels = spectral_clustering(Cosine, n_clusters=cls_num)
    return labels
```

### 2.1

time complexity in secs (60 samples):

<b>Kmeans</b>	<b>0.087019</b>
<b>hierarhical clustering</b>	<b>2.796048</b>
<b>spectral clustering</b>	<b>0.168535</b>

where **Kmeans** is implemented in matrix operation form. Time complexity of hierarhical clustering is largest.

In experiment, **Kmeans** and **spectral clustering** can handle upto 10000 samples, whereas **hierarhical clustering** can only handle upto 100 samples in a reasonable time.

### 2.2

#### Kmeans

$J_e$  and  $NMI$  matches:

```
[1] J_e = 1938321514.0
[1] NMI = 0.438838029906
[2] J_e = 1724484104.0
[2] NMI = 0.486831873336
[3] J_e = 1665295022.0
[3] NMI = 0.508187701536
```

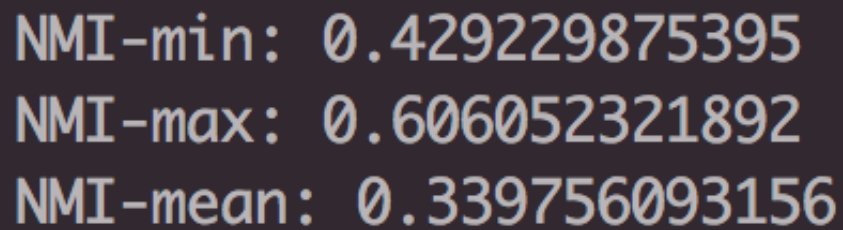
with  $J_e$  decreasing,  $NMI$  increases at the same time.

## Hierarhical clustering

```
start = time.clock()
data_cls = hierarhical(XTrain, YTrain, kind="min")
end = time.clock()
print("hierarhical: {}".format(end-start))
print("NMI-min: {}".format(NMI(data_cls, YTrain)))

data_cls = hierarhical(XTrain, YTrain, kind="max")
print("NMI-max: {}".format(NMI(data_cls, YTrain)))

data_cls = hierarhical(XTrain, YTrain, kind="mean")
print("NMI-mean: {}".format(NMI(data_cls, YTrain)))
```



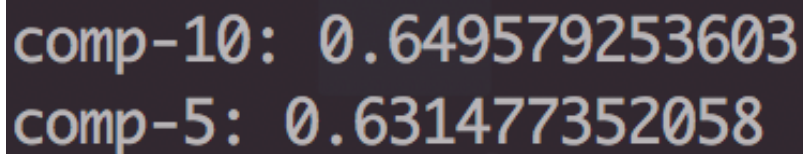
```
NMI-min: 0.429229875395
NMI-max: 0.606052321892
NMI-mean: 0.339756093156
```

max distance yields a best *NMI*

## Spectral clustering

```
start = time.clock()
data_cls = spectral(XTrain, n_components=10)
end = time.clock()
print("spectral: {}".format(end-start))
print("comp-10: {}".format(NMI(data_cls, YTrain)))

data_cls = spectral(XTrain, n_components=5)
print("comp-5: {}".format(NMI(data_cls, YTrain)))
```



```
comp-10: 0.649579253603
comp-5: 0.631477352058
```

a choice of number of eigen vector of 10 result in a best *NMI*

1. Estimate a range of number of clustering by experience, e.g, 1:20
2. Assign the number in this range to each Algorithm, calculate the NMI.
3. Choose the number where NMI increases most steeply.

## 2.4

Considering the trade off between  $NMI$  and time complexity, I might choose the **Kmeans** since its simplicity and efficiency in matrix implementation.