Homework 5: K-means Clustering

Due Tuesday March 5

The OptDigits data from the UCI ML repository will be classified using K-means clustering. Each data instance has 64 attributes (pixels) with a value between 0-16, along with the class label.

Clustering will be evaluated using average mean-square-error, mean- square-separation, mean entropy, and accuracy.

Experiment 1

K = 10

1) Run 5 times with random initial seeds

```
1b) Stop run when clusters stop changing
```

2) Choose run with smallest average MSE

In [137]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math
from scipy import stats
from sklearn.metrics import confusion_matrix
```

In [4]:
```
# Load Data #64 attributes + 1 class = 65
traindata = pd.read_csv("optdigits.train", header = None)
testdata = pd.read_csv("optdigits.test", header = None)

print(traindata.head())
traindata = traindata.values #to numpy matrix
testdata = testdata.values

np.random.seed(0)
```

```
   0   1   2   3   4   5   6   7   8   9  ...  55  56  57  58  59  60
  61  62  \
0  0   1   6  15  12   1   0   0   0   7  ...   0   0   0   6  14   7
   1   0
1  0   0  10  16   6   0   0   0   0   7  ...   0   0   0  10  16  15
   3   0
2  0   0   8  15  16  13   0   0   0   1  ...   0   0   0   9  14   0
   0   0
3  0   0   0   3  11  16   0   0   0   0  ...   0   0   0   0   1  15
   2   0
4  0   0   5  14   4   0   0   0   0   0  ...   0   0   0   4  12  14
   7   0

   63  64
0   0   0
1   0   0
2   0   7
3   0   4
4   0   6

[5 rows x 65 columns]
```

In [90]:
```
def rand_cent(data, k):
    """
    Random datapoint selection for initial centroids
    input:
        data: training data set, to compute length
        k: Number of centroids
    output:
        init_cents: array of k-centroids (k-rows, n_attribute-cols)
    """
    n = len(data) - 1
    init_cents = np.zeros((k, data.shape[1]))
    init_ind = np.random.randint(0, n, k)    #array of random indicie
s into data, of size k
    for i in range(len(init_ind)):
        init_cents[i] = data[init_ind[i]]
    return init_cents

def cluster(data, cents, k):
    """
    Assigns data to clusters
    input:
        data: training data set, to compute euclidian distance (n x 6
4, STRIP CLASS at end first!)
        k: Number of centroids
        cents: array (len: k) of centroids (k x 64 matrix)
    output:
        #dists: k columns of euclidian distance between row = data_po
int, column = centroid
        bins: n- length array mapping data-index to cluster (0 to k-
1)
    """
#     dists = [] #k columns of euclidian distance between row = data_
point, column = centroid
#     for cent in cents:
#         dist = np.linalg.norm(data - cent, axis = 1)
#         dists = np.concatenate((dists, dist.reshape(-1,1)), axis=1)
 if dists.size else dist.reshape(-1,1)
#         #alternative, reshape at end?
#         #dists = np.vstack(dists, dist) if dists.size else dist
#     bins = np.argmin(dists, axis=1)

    dists = np.zeros((k, len(data))) #(k by n_data) of eucl distance
 between row = centroid, col = data_point
    for i in range(len(cents)):
        cent = cents[i]
        dist = np.linalg.norm(data - cent, axis = 1) #1D array of len
gth (data)
        dists[i] = dist
    bins = np.argmin(dists, axis=0)

    return bins

def update(data, bins, k):
    """
    Creates new centroids by averaging clusters
    input:
```

```
        data: training data set, to compute euclidian distance (n x 6
4, STRIP CLASS at end first!)
        bins: n- length array mapping data-index to cluster (0 to k-
1)
        k: Number of centroids
    output:
        new_cents: Array of new centroids (k by ncol(aka n_attribute
s))
    """
    ncol = data.shape[1] #number of columns in data (Attributes)
    csize = np.zeros(k) #track number of data points in each cluster
 for mean calculation
    ctotal = np.zeros((k, ncol)) #k rows, each with ncol (64) attribu
tes

    for i in range(len(data)):
        ctotal[bins[i]] += data[i]
        csize[bins[i]] += 1

    csize[csize == 0] = 1 #for divide by 0 cases
    new_cents = ctotal / csize.reshape(-1,1)
    return new_cents

def run_mult(data, k, n_runs):
    """
    Runs the clustering until the centers no longer change, for n_run
s.
    input:
        data: training data set, to compute euclidian distance (n x 6
4, STRIP CLASS at end first!)
        k: Number of centroids
    output:
        Array of trained centroids (k by ncol(aka n_attributes) by n_
runs)

    """
    final_cents = np.zeros((k, data.shape[1], n_runs)) #3D array with
 k by n_attributes by height n_runs

    for i in range(n_runs):

        new_cents = rand_cent(data , k)
        counter = 0
        while(True):
            cents = new_cents
            bins = cluster(data, cents, k)
        #     print(bins.shape)
        #     print(bins[0:100])
            new_cents = update(data, bins, k)
            counter += 1

            if np.array_equal(cents, new_cents):
                break
            elif counter > 100:
                print("max iterations exceeded, 100")
                break
        print("iterations: " + str(counter))
```

```
        final_cents[:,:,i] = cents
    return final_cents
```

In [89]:
```
k = 10
train = traindata[:,:-1] #without labels

cents_5 = run_mult(train, k, 5)
```

```
iterations: 31
iterations: 32
iterations: 26
iterations: 41
iterations: 24
```

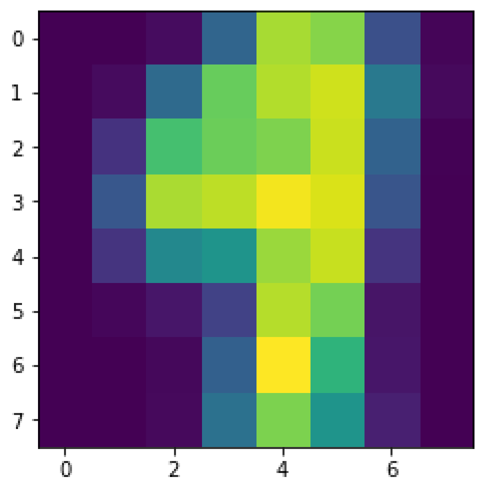
In [148]:
```
def avg_mse(data, cents, k):
    """
    Average mean square error
    input:
        data: training data set, to compute euclidian distance (n x 6
4, STRIP CLASS at end first!)
        cents (k by ncol(aka n_attributes))
        k: num clusters
    output:
        ret: avg MSE
    """
    mses = []
    bins = cluster(data, cents, k)
    data = np.concatenate((data, bins.reshape(-1,1)), axis=1) #concat
 bin column to the right side (65th column)

    for i in range(len(cents)):
        if np.sum(cents[i]) == 0:
            continue
        cent_data = data[data[:,64]==i][:,:-1] #data with bin = i
        dist = np.linalg.norm(cent_data - cents[i], axis = 1)
        mse = np.mean(dist**2) #average dist^2
        mses += [mse]
    return np.mean(mses)

def mss(cents, k):
    """
    Mean Square Separation
    input:
        cents (k by ncol(aka n_attributes))
        k: num clusters
    output:
        ret: MSS
    """
    total = [] #Array of d(mu.i, mu.j)^2, aka running total to be ave
raged later
    for i in range(k):
        for j in range(i+1,k):
            dist = np.linalg.norm(cents[i] - cents[j])
            total += [dist**2] #add dist squared
    return np.mean(total)

def m_entropy(train, cents, k):
    """
    Mean Entropy
    input:
        train: training data set, (n x 65, WITH CLASS AT END)
        test: test data, to classify
        cents (k by ncol(aka n_attributes))
        k: num clusters
    output:
        ret: Mean entropy
    """
    total = 0 #Running totl of entropy

    data_train = train[:,:-1] #strip label
```

```
    bins = cluster(data_train, cents, k) #predicted cluster
    label = train[:,64] #provided class label
    bincount = [len(bins[bins[:] == x ]) for x in range(k)] #number o
f instances in each bin
    bintotal = len(bins)
    print(bincount, bintotal)

    #sum weighted(mean)-entropy per cluster
    for i in range(k):
        w = bincount[i]/bintotal #instances in cluster i / total inst
ances
        entropy = 0
        for j in range(k): #instances in cluster i, that belong to cl
ass j
            numj = sum(label[bins[:]==i] == j)
            probj = numj / bincount[i]
            if numj == 0:
                continue
            entropy += -(probj * math.log(probj, 2))
        total += w*entropy

    return total

#Drawing function
def draw_digit(data):
    """
    input:
        vector of grayscaled pixel values
    output: image of (sqrt(vector) by sqrt(vector) pixels)
    """
    size = int(len(data)**0.5)
    print(size," by ", size)
    img = np.reshape(data, (size, size))
    #print(img)
    pic = plt.imshow(img)
    plt.show(pic)
    return

def classify(train, test, cents, k):
    """
    Associates each cluster with the most frequent class contained wi
thin.
    Assign each test point the class of the nearest cluster.
    input:
        train: training data set, (n x 65, WITH CLASS AT END)
        test: test data, to classify
        cents (k by ncol(aka n_attributes))
        k: num clusters
    output:
        pred: predicted classes of the test_data
        modes: cluster modes
    """
    data_train = train[:,:-1] #strip label
    bins = cluster(data_train, cents, k) #predicted label
    #data_train = np.concatenate((data_train, bins.reshape(-1,1)), ax
is=1) #concat bin column to the right side (65th column)
    modes = np.zeros(k)
```

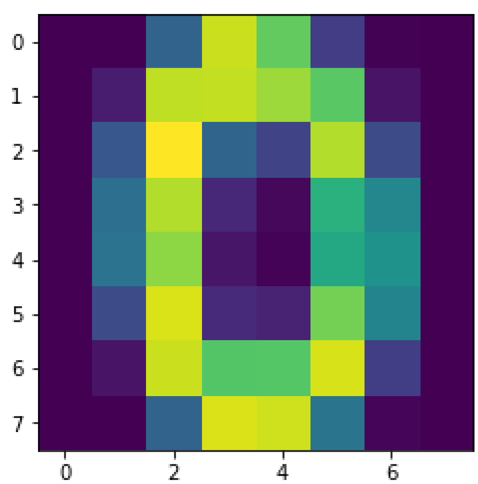
```
    pred = np.zeros(len(test)) #Predicted classes of test data

    for i in range(len(cents)):
        if np.sum(cents[i]) == 0:
            continue
        #train_cent_i = train[data_train[:,64]==i] #all training data
 labels, of cluster = i
        train_cent_i = train[bins==i]
        real_labels = train_cent_i[:,64]
        m = stats.mode(real_labels)
#         print("m")
#         print(m[0])
        modes[i] = m[0]

    data_test = test[:,:-1] #strip label
    bins_test = cluster(data_test, cents, k)
    pred = [modes[x] for x in bins_test]

    print("modes", modes)
    return pred, modes

#****** CLASSIFICATION and METRICS *******************
#MSE
avg_mses = []
for i in range(cents_5.shape[2]):
    avg_mses += [avg_mse(train, cents_5[:,:,i], k)]

best_ind = np.argmin(avg_mses)
best_cents = cents_5[:,:,best_ind]

print("avg mses:", avg_mses)
print("best avg_mse = ",best_ind,"@ ", avg_mses[best_ind])

#Mean Square Separation
my_mss = mss(best_cents, k)
print("mss: ", my_mss)

#Mean Entropy
my_ent = m_entropy(traindata, best_cents, k)
print("mean entropy: ", my_ent)

#Classify
my_pred, my_modes = classify(traindata,testdata,best_cents,k)

#Accuracy
accuracy = np.sum(my_pred == testdata[:,64]) / len(my_pred)
print("accuracy = " + str(accuracy))


#Confusion matrix
cm = confusion_matrix(testdata[:,64], my_pred)
print("confusion matrix (digits 0-9)")
print(cm)

#Visualize non-empty clusters
for i in range(len(best_cents)):
    if sum(best_cents[i]) == 0:
```

```
        continue
    print("cluster ", i, " mode = ", my_modes[i])
    draw_digit(best_cents[i])
```

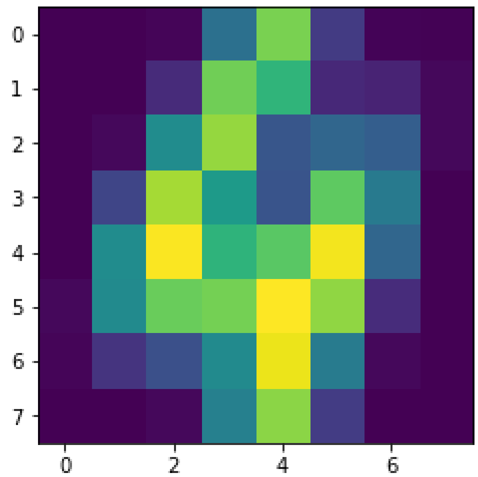
```
avg mses: [643.3606393787893, 647.5086661650396, 643.4561750961491, 6
43.4322835539555, 647.6597981670526]
best avg_mse =  0 @  643.3606393787893
mss:  1302.3024352266348
[283, 372, 312, 458, 530, 781, 167, 301, 229, 390] 3823
mean entropy:  0.9671064023612094
modes [1. 0. 4. 7. 8. 3. 2. 5. 2. 6.]
accuracy = 0.7417918753478019
confusion matrix (digits 0-9)
[[176   0   0   0   2   0   0   0   0   0]
 [  0  56  23   1   0   0   2   0 100   0]
 [  1   1 162   0   0   0   0   2  11   0]
 [  0   0   2 163   0   1   0   9   8   0]
 [  0   5   0   0 162   0   0   6   8   0]
 [  0   1   0  31   1 148   1   0   0   0]
 [  1   0   0   0   1   0 176   0   3   0]
 [  0   7   0   0   1   1   0 168   2   0]
 [  1   8   1  34   0   4   2   2 122   0]
 [  0  23   0 145   0   6   0   5   1   0]]
cluster  0  mode =  1.0
8  by  8
```



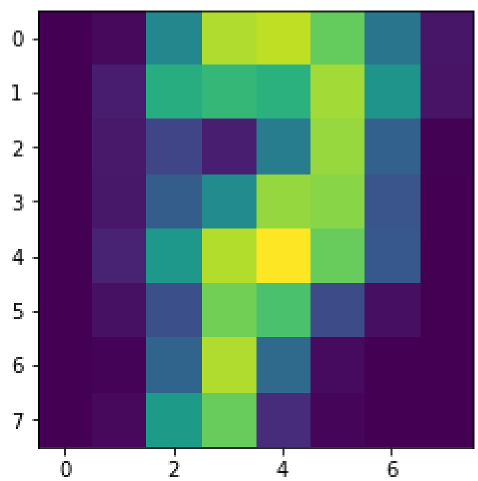
```
cluster  1  mode =  0.0
8  by  8
```



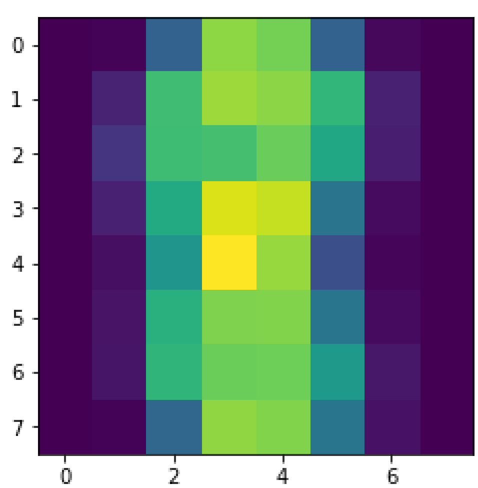
```
cluster  2  mode =  4.0
8  by  8
```



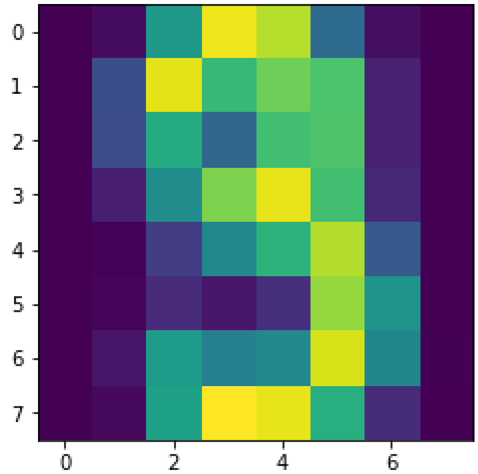
```
cluster  3  mode =  7.0
8  by  8
```



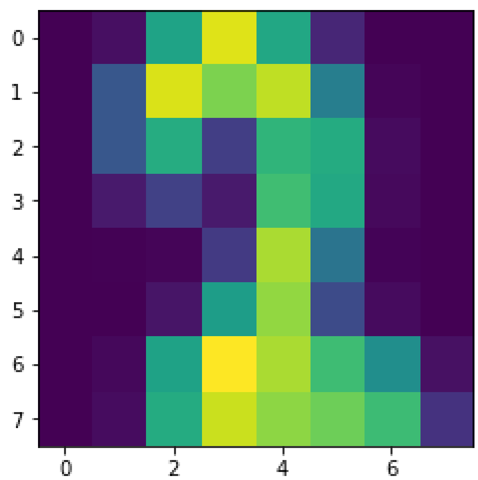
```
cluster  4  mode =  8.0
8  by  8
```



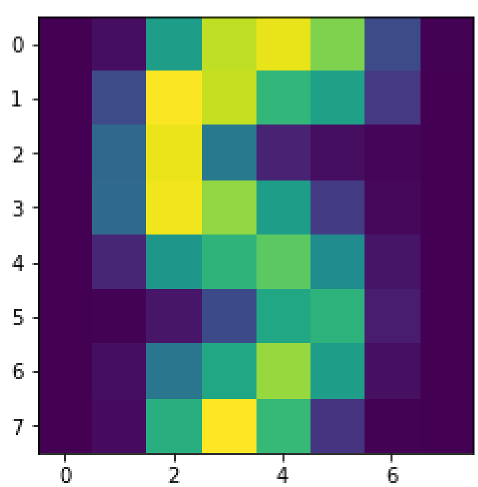
```
cluster  5  mode =  3.0
8  by  8
```



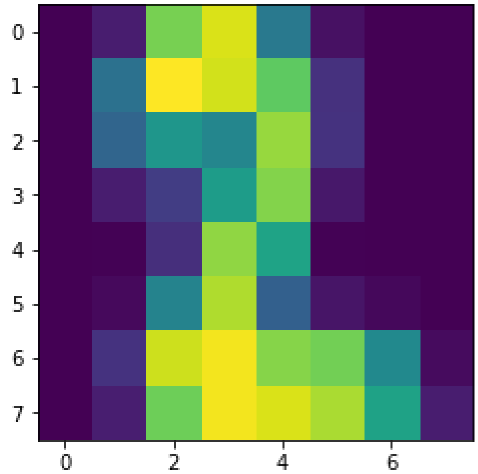
```
cluster  6  mode =  2.0
8  by  8
```



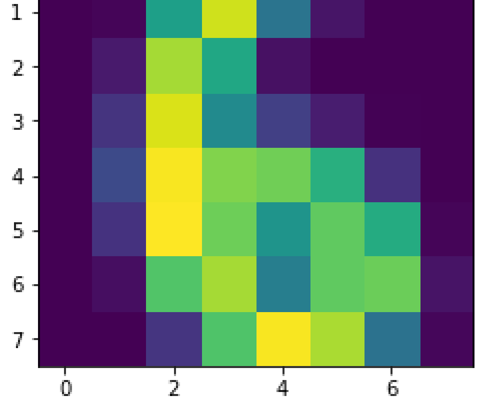
```
cluster  7  mode =  5.0
8  by  8
```



```
cluster  8  mode =  2.0
8  by  8
```




```
cluster  9  mode =  6.0
8  by  8
```



Experiment 1 Discussion

The best average MSE was 643, MSS was 1302, and mean entropy was 0.967. Accuracy on the test data was 0.74 -- a decent result given we have ten classes.

We see in the modes that the digit "2" is represented twice, but at the cost of the digit "9" not being represented at all. This is reflected in the confusion matrix (zero 9's predicted). Using the cluster average values, we can visualize each cluster as seen above, and for the most part we can match the images with their associated cluster modes.

Interestingly enough, the accuracy was higher when choosing the best run based on Mean-error, rather than mean-square-error when we forgot to square the distance in the MSE function originally. This may suggest a couple of outliers cause the best MSE to differ from the best ME, as squaring accentuates points further from their cluster centers. The accuracy was ~79% when choosing run_index = 3 (whereas the MSE chose run_index = 0). We will note that index=3 was the second best still when using squared-error.

Experiment 2

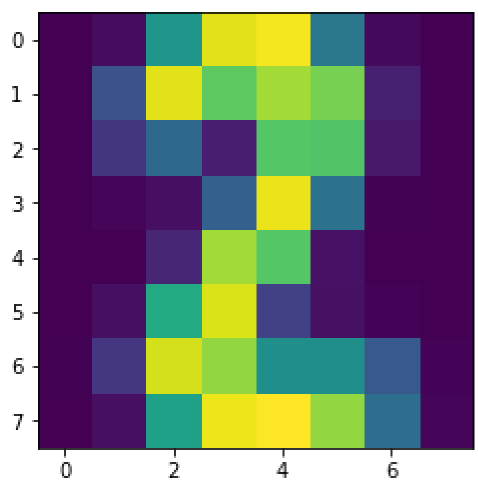
We repeat experiment 1, except with 30 clusters. k = 30

In [149]:
```
k2 = 30
train = traindata[:,:-1] #without labels

cents_5_e2 = run_mult(train, k2, 5)
```

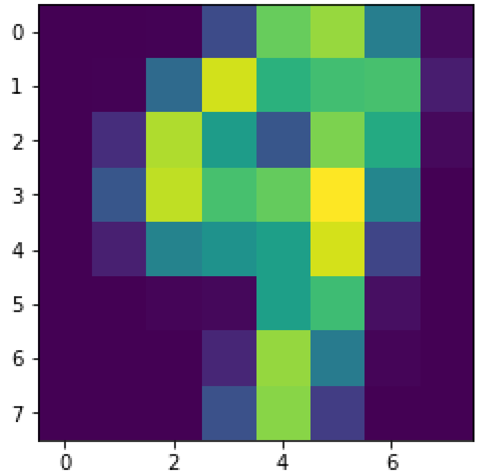
```
iterations: 29
iterations: 29
iterations: 33
iterations: 26
iterations: 30
```

In [151]:
```
#****** CLASSIFICATION and METRICS *******************
#MSE
avg_mses = []
for i in range(cents_5_e2.shape[2]):
    avg_mses += [avg_mse(train, cents_5_e2[:,:,i], k2)]

best_ind = np.argmin(avg_mses)
best_cents = cents_5_e2[:,:,best_ind]

print("avg mses:", avg_mses)
print("best avg_mse = ",best_ind,"@ ", avg_mses[best_ind])

#Mean Square Separation
my_mss = mss(best_cents, k2)
print("mss: ", my_mss)

#Mean Entropy
my_ent = m_entropy(traindata, best_cents, k2)
print("mean entropy: ", my_ent)

#Classify
my_pred, my_modes = classify(traindata,testdata,best_cents,k2)

#Accuracy
accuracy = np.sum(my_pred == testdata[:,64]) / len(my_pred)
print("accuracy = " + str(accuracy))


#Confusion matrix
cm = confusion_matrix(testdata[:,64], my_pred)
print("confusion matrix (digits 0-9)")
print(cm)

#Visualize non-empty clusters
for i in range(len(best_cents)):
    if sum(best_cents[i]) == 0:
        continue
    print("cluster ", i, " mode = ", my_modes[i])
    draw_digit(best_cents[i])
```

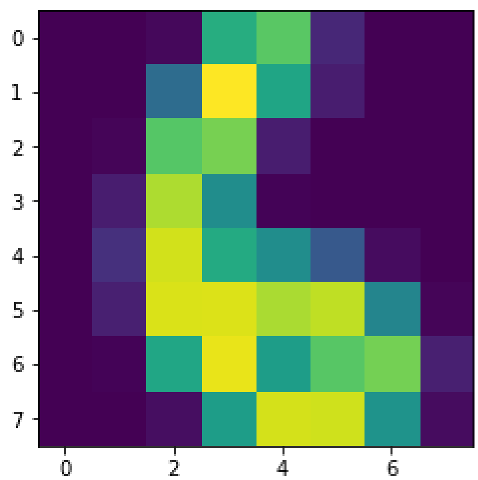
```
avg mses: [483.88289373479336, 478.2082203520066, 491.0342002562123,
479.78682583118916, 477.0359078266181]
best avg_mse =  4 @  477.0359078266181
mss:  1550.3181302871806
[73, 63, 115, 188, 194, 90, 131, 97, 65, 186, 125, 143, 68, 89, 127,
109, 103, 95, 136, 222, 100, 107, 90, 84, 161, 101, 102, 152, 180, 32
7] 3823
mean entropy:  0.3533753677463645
modes [2. 9. 6. 7. 0. 1. 1. 9. 4. 7. 5. 9. 6. 2. 5. 8. 1. 2. 2. 8. 5.
4. 1. 4.
 9. 6. 6. 4. 0. 3.]
accuracy = 0.9087367835281024
confusion matrix (digits 0-9)
[[177   0   0   0   1   0   0   0   0   0]
 [  0 162  13   1   0   1   2   0   0   3]
 [  0   1 174   0   0   0   0   2   0   0]
 [  0   0   2 136   0   3   0   6  10  26]
 [  0   6   0   0 173   0   1   0   0   1]
 [  0   0   0   0   1 174   0   0   0   7]
 [  0   3   0   0   0   1 175   0   2   0]
 [  0   0   0   0   7   0   0 159   2  11]
 [  0  20   1   1   2   2   1   1 137   9]
 [  1   0   0   4   7   1   0   0   1 166]]
cluster  0  mode =  2.0
8  by  8
```



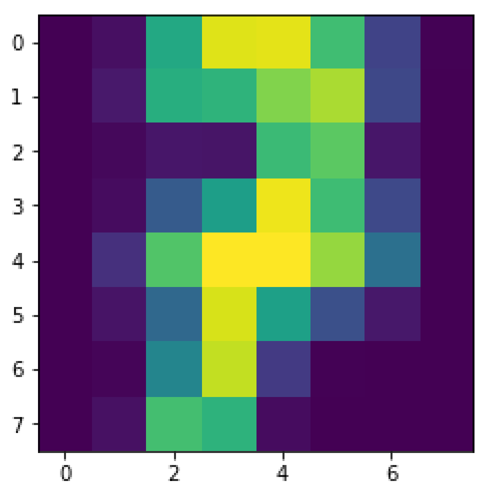

```
cluster  1  mode =  9.0
8  by  8
```



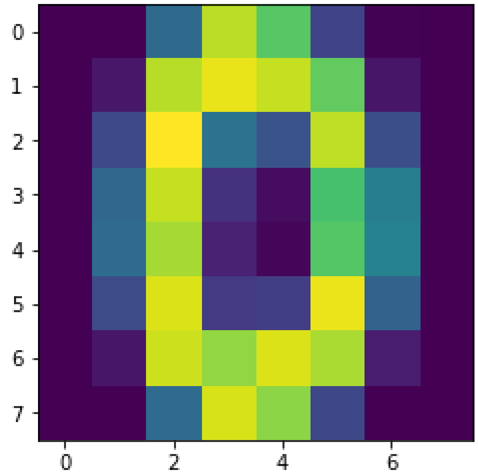
cluster 2 mode = 6.0
8 by 8



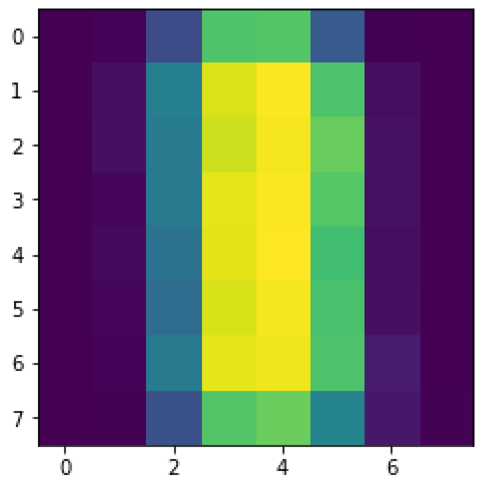
cluster 3 mode = 7.0
8 by 8



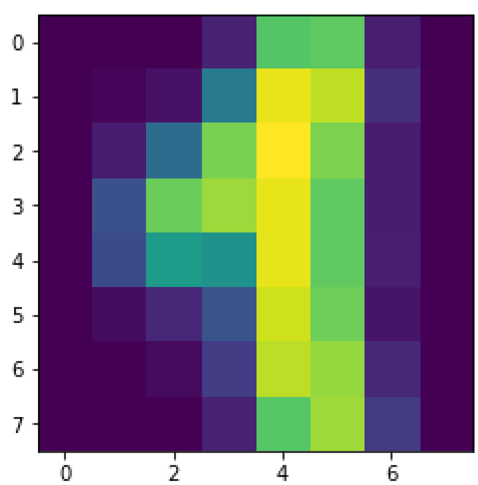
cluster 4 mode = 0.0
8 by 8



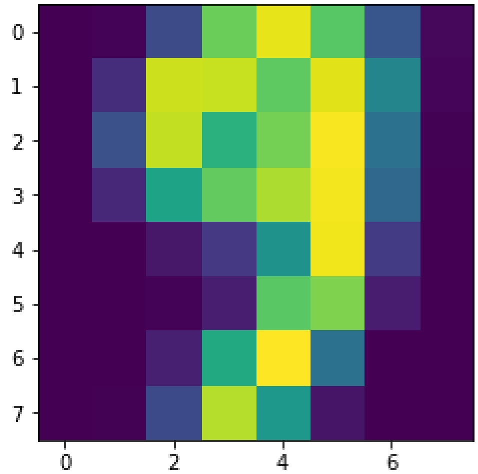
cluster 5 mode = 1.0
8 by 8



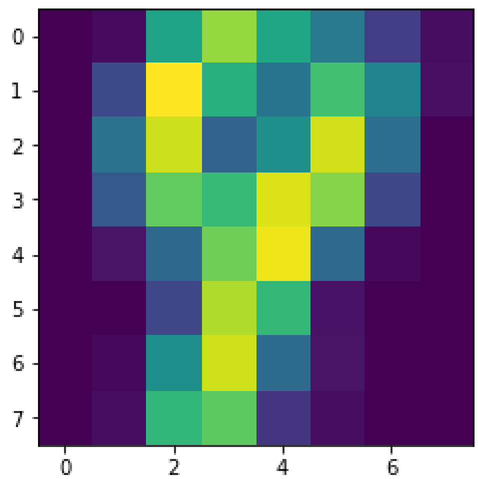
cluster 6 mode = 1.0
8 by 8



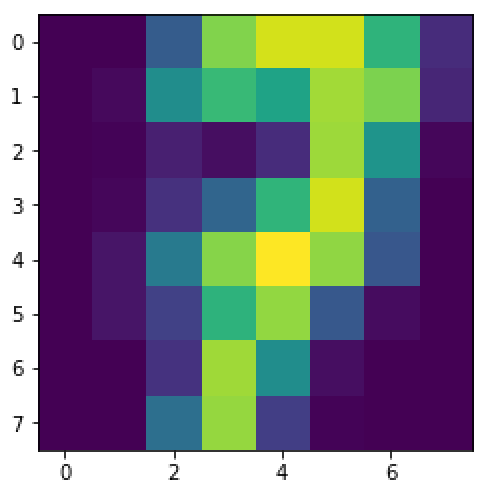
cluster 7 mode = 9.0
8 by 8



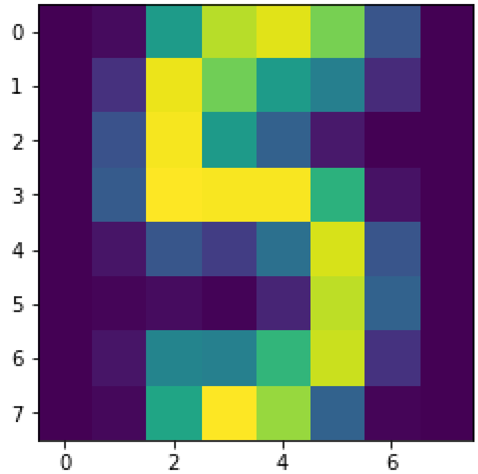
```
cluster  8  mode =  4.0
8  by  8
```



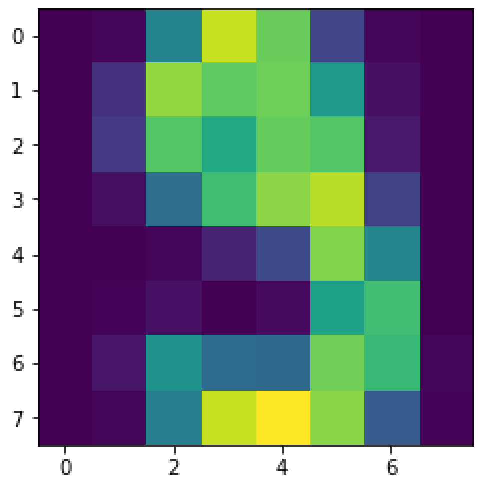
```
cluster  9  mode =  7.0
8  by  8
```



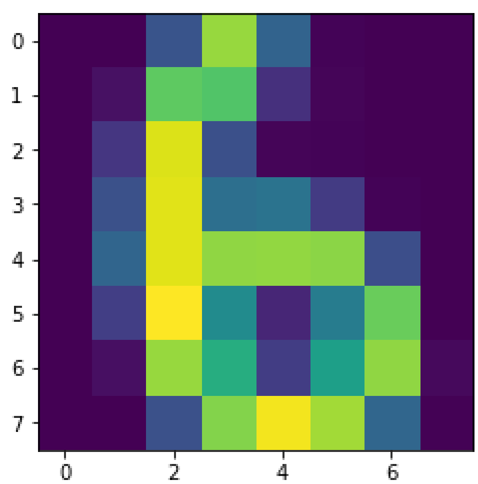
```
cluster  10  mode =  5.0
8  by  8
```



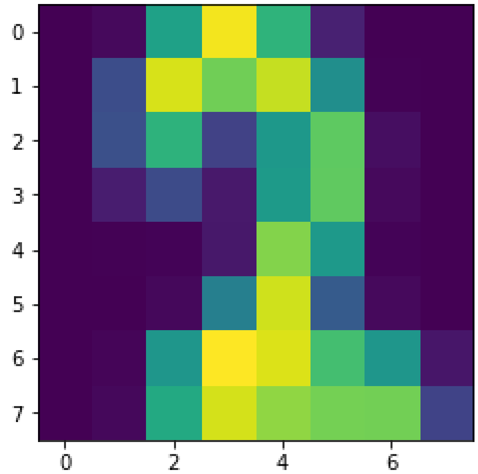
```
cluster  11  mode =  9.0
8  by  8
```



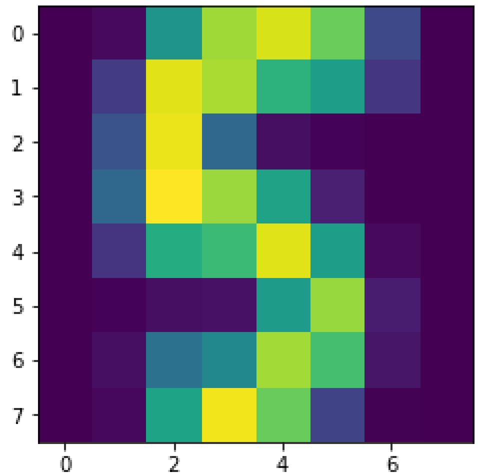
```
cluster  12  mode =  6.0
8  by  8
```



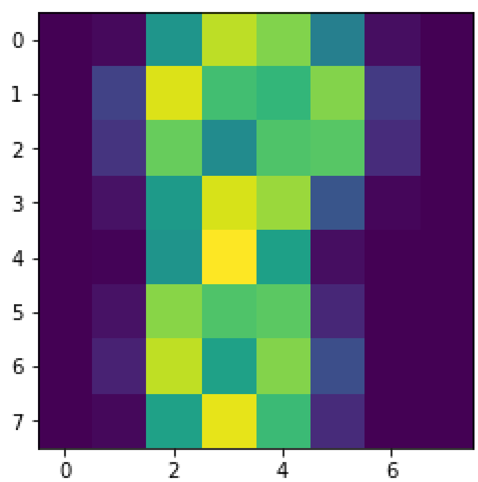
```
cluster  13  mode =  2.0
8  by  8
```



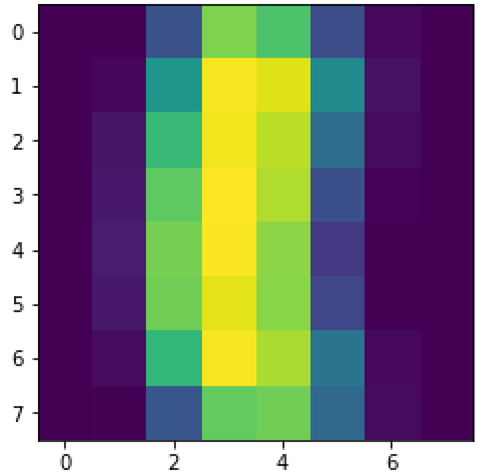

```
cluster  14  mode =  5.0
8  by  8
```



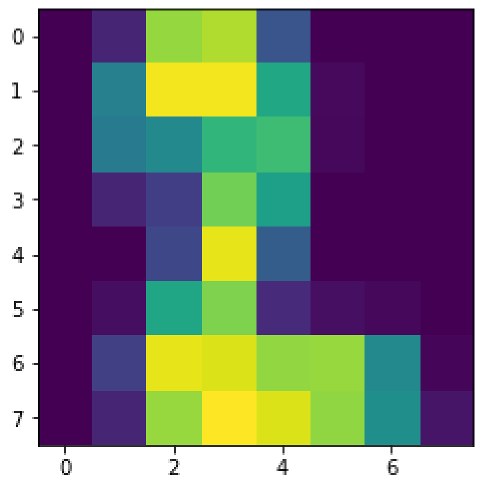

```
cluster  15  mode =  8.0
8  by  8
```



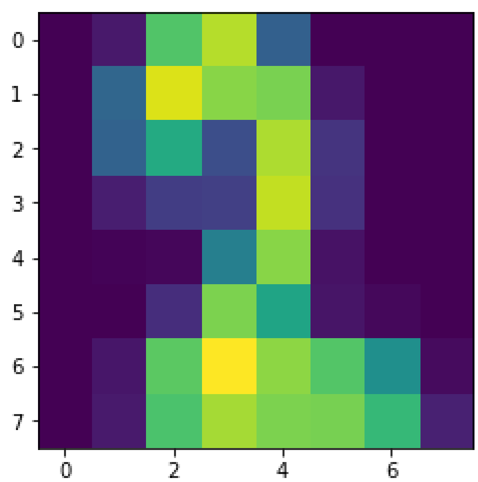

```
cluster  16  mode =  1.0
8  by  8
```



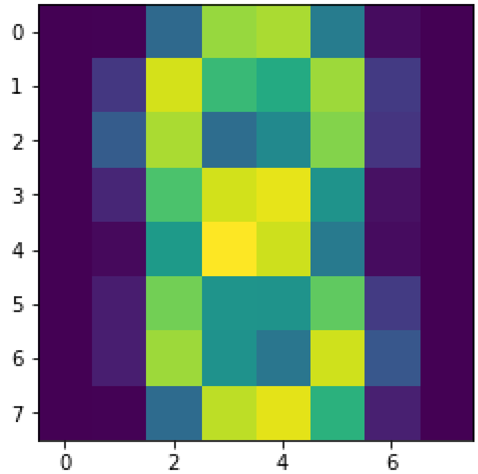

```
cluster  17  mode =  2.0
8  by  8
```



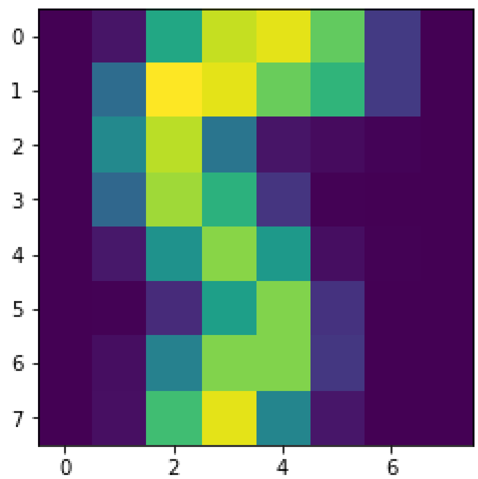

```
cluster  18  mode =  2.0
8  by  8
```



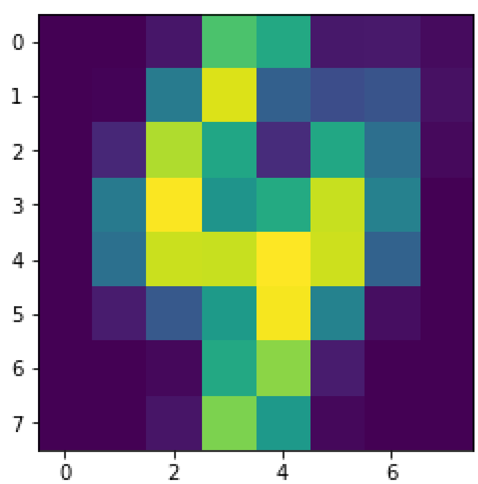

```
cluster  19  mode =  8.0
8  by  8
```



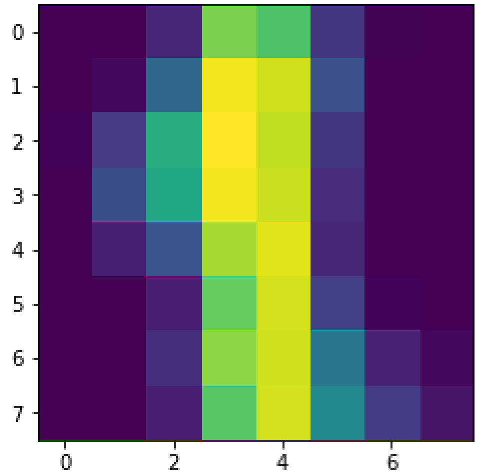
cluster 20 mode = 5.0
8 by 8



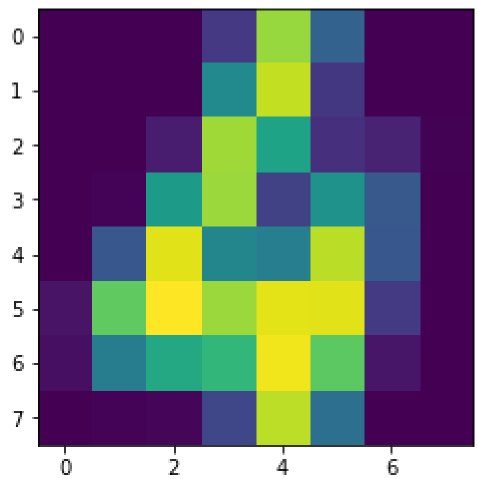
cluster 21 mode = 4.0
8 by 8



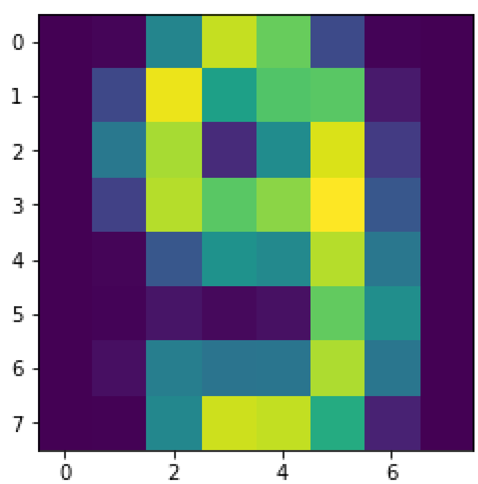
cluster 22 mode = 1.0
8 by 8



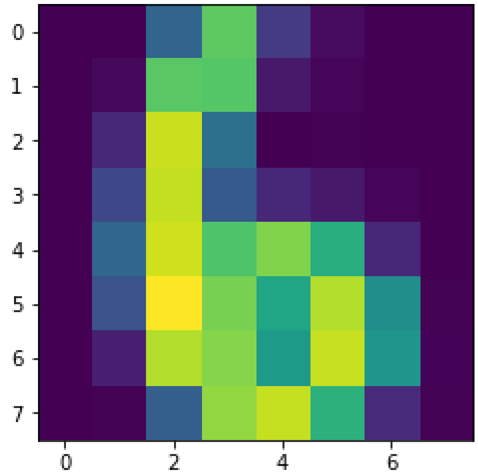
cluster 23 mode = 4.0
8 by 8



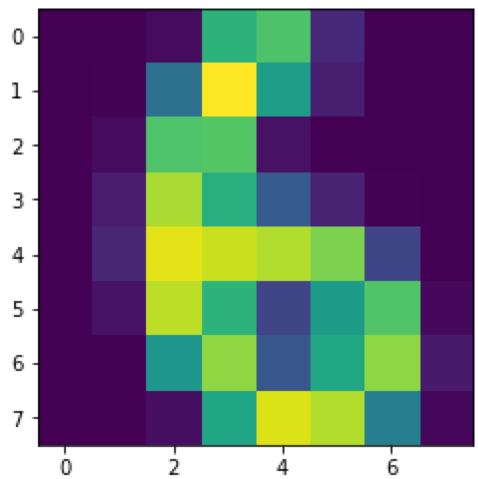
cluster 24 mode = 9.0
8 by 8



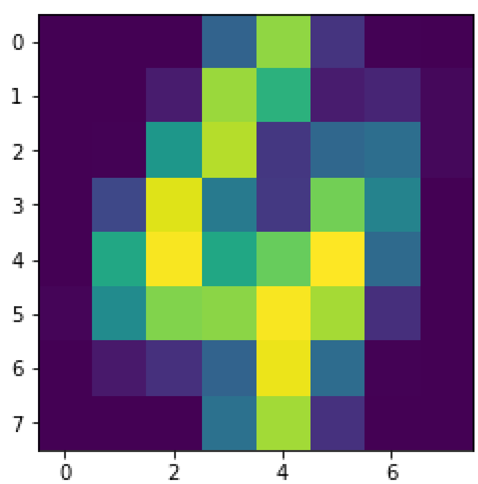
cluster 25 mode = 6.0
8 by 8



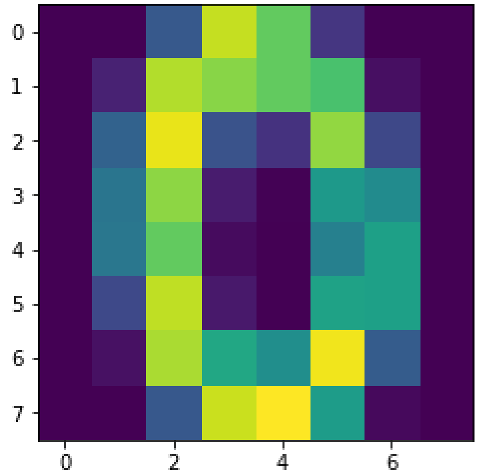

```
cluster  26  mode =  6.0
8  by  8
```



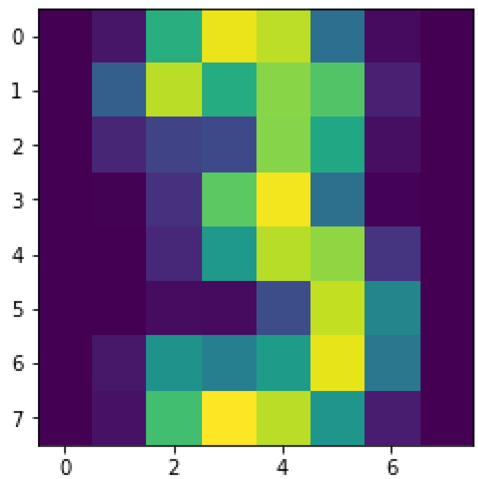

```
cluster  27  mode =  4.0
8  by  8
```




```
cluster  28  mode =  0.0
8  by  8
```




```
cluster  29  mode =  3.0
8  by  8
```



Experiment 2 Discussion

The best average MSE was 477, MSS was 1550, and mean entropy was 0.353. Accuracy on the test data was 0.908. All metrics show significant improvement with 30 clusters vs experiment 1's 10 clusters. MSE dropped (643 to 477), MSS increased (1302 to 1550), and entropy dropped (~1 to 0.35). We can safely conclude that 30 clusters outperformed the 10 clusters, with a significant accuracy jump from 0.75 to 0.91.

We see in the modes that with more clusters, all digits are represented as least once. This is reflected in the confusion matrix, as we no longer have any 0's in the diagonal. In fact, all digits have multiple clusters assigned to them, with the lone exception of "3", which only got a single cluster. This could suggest the number "3" is the most distinct, with fewer variations in writing styles, whereas we can see in the visualization for clusters 5 & 6 that the digit "1" can be written either as a solid vertical line (cluster 5), or as a line with the top part bent (cluster 6). The visualizations seem slightly clearer as well, as a result of the lower average entropy.

In [79]:
```
#MISC TESTING GROUND AND RESOURCES USED

a = np.array([[1,2,3],[2,3,4],[3,4,5]])
print(a)
b = np.array([1,5,2])
print(b)
c = a-b
print(c)
#https://stackoverflow.com/questions/7741878/how-to-apply-numpy-linal
g-norm-to-each-row-of-a-matrix
#https://docs.scipy.org/doc/numpy/reference/generated/numpy.concatena
te.html
print(np.linalg.norm(c, axis = 1))
print(b.reshape(-1,1))
print(np.concatenate((a, b.reshape(-1,1)), axis=1))
print(np.vstack([a,b]))

#https://stackoverflow.com/questions/22732589/concatenating-empty-arr
ay-in-numpy
#e = []
#print(np.concatenate((e,a), axis = 1))

print(np.argmin(a, axis=1))

print(b.shape[0])

a[0] += [10, 0, 10]
print(a)
print(a/np.array([10, 2, 1]).reshape(-1,1))
print(a[:,:-1])
```

```
[[1 2 3]
 [2 3 4]
 [3 4 5]]
[1 5 2]
[[ 0 -3  1]
 [ 1 -2  2]
 [ 2 -1  3]]
[3.16227766 3.         3.74165739]
[[1]
 [5]
 [2]]
[[1 2 3 1]
 [2 3 4 5]
 [3 4 5 2]]
[[1 2 3]
 [2 3 4]
 [3 4 5]
 [1 5 2]]
[0 0 0]
3
[[11  2 13]
 [ 2  3  4]
 [ 3  4  5]]
[[1.1 0.2 1.3]
 [1.  1.5 2. ]
 [3.  4.  5. ]]
[[11  2]
 [ 2  3]
 [ 3  4]]
```

In [140]:
```
#Array deep copy check
a = np.array([1, 2])
b = a
print(b)
a = np.array([2, 3])
print(b)

b[b==2] = 0
print(b)
#Select rows based on val
c = np.array([[1,2],[2,3],[4,2]])
print(c)
d = c[c[:,1]==2]
print("select rows by val", d)

e = []
e += [1]
e += [2]
print(e)
print(c.shape[0])

#mode
#https://stackoverflow.com/questions/16330831/most-efficient-way-to-f
ind-mode-in-numpy-array


#Array A1 contains indicies into array A2
#List comprehension
a1 = np.array([1, 5, 1, 6])
a2 = np.array([10, 11, 12, 13, 14, 15, 16])
b1 = [a2[x] for x in a1]
print("list comprehension", b1)

#dist calc
c1 = np.array([5, 5])
c2 = np.array([10, 11])
d1 = np.linalg.norm(c1-c2)
print("norm", d1)
print(a1**2)
print(sum(a1==1))
```

```
[1 2]
[1 2]
[1 0]
[[1 2]
 [2 3]
 [4 2]]
select rows by val [[1 2]
 [4 2]]
[1, 2]
3
list comprehension [11, 15, 11, 16]
norm 7.810249675906654
[ 1 25  1 36]
2
```

In [ ]: