ECE174 MiniProject 1

Problem 1: Least Squares Classifier ¶

Kwok Hung Ho A15151703

In [109]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
import numpy as geek
X = loadmat("mnist.mat")
data_train = X['trainX'] # Training set digits
data_label = X['trainY'] # Training set labels
data_test = X['testX'] # Training set digits
data_test_label = X['testY']
inv   = np.linalg.inv
pinv  = np.linalg.pinv
solve = np.linalg.solve
rank  = np.linalg.matrix_rank
norm  = np.linalg.norm
det   = np.linalg.det
solve = np.linalg.solve
lstsq = np.linalg.lstsq
```

All the functions used are declared below:

In [208]:

```
def onevsall(data, label):
    H = []
    G = []
    theta = np.array([])
    aggG =  np.array([])
    for t in range(10):
        G=[]
        H=[]
        for i in range(len(data)):
            if (label[0,i] == t):
                H.append(data[i,:])
                G.append(1)
            else:
                H.append(data[i,:])
                G.append(-1)
        Gnp = np.array(G)
        aggG = np.append(aggG, Gnp)
        beta = lstsq(data, Gnp, rcond=None)[0]
        theta = np.append(theta,beta)
    theta = theta.reshape(10, 784)
    aggG = aggG.reshape(len(data),10)
    return theta, aggG

def onevsallconfusionmatrix(data, label, theta):
    confusion = np.zeros(100).reshape(10,10)
    #theta, aggG = onevsall(data, label)
    raw_label = data@(np.transpose(theta))
    a=[]
    predictlabel = []
    for i in range(len(data)):
        a = np.argmax(raw_label[i,:])
        b = label[:,i]
        confusion[a,b] = confusion[a,b]+1
        confusion = confusion.astype(int)
        predictlabel.append(a)
    print(confusion)
    return confusion, predictlabel

def onevsone(data, label):
    #TO TEST USE H[i]@theta[i]=aggG[i] i from 0 to 44d
    #aggH and aggG are lists of numpy arrrays
    theta = np.array([])
    aggH = []
    aggG =  []
    #thetadict = {}
    #labeldict = {}
    for i in range(10):
        for j in range(10):
            if i<j:
                H = []
                G = []
                for x in range(len(data)):
                    if (data_label[0,x] == i):
                        H.append(data[x,:])
                        G.append(1)
                    elif (label[0,x] == j):
                        H.append(data[x,:])
                        G.append(-1)
                H = np.array(H)
```

```
                aggH.append(H)
                G = np.array(G)
                aggG.append(G)
                beta = lstsq(H, G, rcond=None)[0]
                theta = np.append(theta,beta)
                #theta = np.transpose(theta)
                #print(aggG.shape)
                #print(Gnp.shape)
                #print(theta.shape)
                #print(H.shape)
    theta = theta.reshape(45,784)
    theta = theta.T
    return theta, aggG, aggH

def onevsone45mapping():
    main_array = np.empty((0,9), int)
    init = 9
    start_val = 9
    for i in range(9):
        vals = 9 - i
        #for j in range(vals):
        zeroes = np.zeros(i)
        if i == 0:
            zeroes = np.zeros(0)
            values = np.arange(0, 9)
            init = init - 1
        else:
            values = np.arange(start_val, start_val + vals)
            start_val = start_val + init
            init = init - 1
        append_array = np.hstack([zeroes, values])
        #print(append_array)
        main_array = np.vstack([main_array, append_array])
        #print(main_array)
    return main_array

def onevsoneconfusionmatrix(data, label, theta): #theta should be obtained from data_tr
ain
    thetaij = onevsone45mapping() # maps 0-1, 0-2,...8-9, t0 0,, 1,...44
    A = geek.sign(data@theta)
    votetable = np.zeros((len(data[:,0]),10))
    predlabel=[]
    for i in range(len(data[:,0])): # making the vote table
        for j in range(45):
            if A[i,j] ==1:
                arg = np.argwhere(thetaij==j)[0][0]
                votetable[i,arg] +=1
            elif A[i,j] ==-1:
                arg = np.argwhere(thetaij==j)[0][1]
                votetable[i,arg+1] +=1
        predlabel.append(np.argmax(votetable[i]))

    confusion = np.zeros((10,10)) # Making the confusion matrix
    for i in range(len(data[:,0])):
        a = predlabel[i]
        b = label[:,i]
        confusion[a,b] = confusion[a,b]+1
        confusion = confusion.astype(int)
    #print(confusion)
    return predlabel, thetaij, confusion
```

One Vs One

Obtained weights via least squares with training data and tested it on the test data. Results are below:

In [185]:

```
theta, aggG, aggH = onevsone(data_train, data_label)
predlabel, thetaij, confusion = onevsoneconfusionmatrix(data_test, data_test_label, the
ta)
print(thetaij)
print(confusion)
```

```
[[ 0.  1.  2.  3.  4.  5.  6.  7.  8.]
 [ 0.  9. 10. 11. 12. 13. 14. 15. 16.]
 [ 0.  0. 17. 18. 19. 20. 21. 22. 23.]
 [ 0.  0.  0. 24. 25. 26. 27. 28. 29.]
 [ 0.  0.  0.  0. 30. 31. 32. 33. 34.]
 [ 0.  0.  0.  0.  0. 35. 36. 37. 38.]
 [ 0.  0.  0.  0.  0.  0. 39. 40. 41.]
 [ 0.  0.  0.  0.  0.  0.  0. 42. 43.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 44.]]
[[ 959    0    9    8    2    8    8    1    8    6]
 [   0 1122   20    2    3    7    5   18   16    5]
 [   2    3  931   14    6    3    8   16    7    1]
 [   1    3   12  930    1   31    0    3   20   12]
 [   0    0   11    1  926    8    5   11   11   29]
 [   4    1    3   21    1  788   15    1   38   10]
 [   8    3   12    2    7   18  915    0    9    0]
 [   3    1    8    8    4    2    0  954   10   22]
 [   2    2   26   20    4   22    2    2  845    5]
 [   1    0    0    4   28    5    0   22   10  919]]
```

The first matrix is the mapping matrix for the one vs one classifier. Using this matrix, I can correctly increment the votetable for each picture, thus showing how many votes are cast for each class. After that, I obtained the predicted labels through the argument in the votetable with the highest votes. The confusion matrix is then obtained by comparing the predicted label and the actual label.

Below is the confusion matrix obtained from running it on the training data.

In [207]:

```
theta, aggG, aggH = onevsone(data_train, data_label)
predlabel, thetaij, confusion = onevsoneconfusionmatrix(data_train, data_label, theta)
print(confusion)
finderror(predlabel, data_label)
```

```
[[5811    2   52   27   14   46   27   11   33   22]
 [   2 6633   64   41   20   45   19   65  194   17]
 [  14   28 5518  110   19   35   29   56   49   16]
 [   8   22   62 5619    3  160    1    6  107   84]
 [  11    6   54    9 5570   26   27   71   44  144]
 [  21   10   20  122    9 4909   79    9  155   27]
 [  22    2   45   22   21  106 5704    1   36    3]
 [   4    8   36   48   17    7    0 5874   26  130]
 [  29   23   92   91    6   67   31    8 5147   37]
 [   1    8   15   42  163   20    1  164   60 5469]]
number of errors: 3746
error rate is: 0.062433333333333334
```

Analyzing the error:

This error is for theta found from the training set applied to the test set. (where theta is the weights)

In [175]:

```
predlabel, thetaij, confusion = onevsoneconfusionmatrix(data_test, data_test_label, the
ta)
finderror(predlabel, data_test_label)
```

```
number of errors: 711
error rate is: 0.0711
```

The error seen above is about 0.0711, corresponding to 7.1%. The error represents the results of the weights obtained from training data on the test data.

One Vs All

In [205]:

```
theta, aggG = onevsall(data_train, data_label)
confusion, predictlabel = onevsallconfusionmatrix(data_train, data_label, theta)
finderror(predictlabel, data_label)
```

```
[[5665    1   93   42    9  144  107   53   82   65]
 [   8 6514  258  142   97   70   66  187  527   60]
 [  19   36 4802  183   56   30   65   43   58   25]
 [  18   16  149 5206    8  520    1   57  223  113]
 [  26   10  101   29 5125   82   59  153  123  355]
 [  43   29   11   92   49 3786   81    9  234    8]
 [  73   15  237   58   48  196 5491    3   50    4]
 [   3   13   84  109   23   38    0 5391   21  488]
 [  62  103  203  139   83  404   46   18 4358   53]
 [   6    5   20  131  344  151    2  351  175 4778]]
number of errors: 8884
error rate is: 0.14806666666666668
```

The confusion matrix above is for the training data.

In [206]:

```
theta, aggG = onevsall(data_train, data_label)
confusion, predictlabel = onevsallconfusionmatrix(data_test, data_test_label, theta)
finderror(predictlabel, data_test_label)
```

```
[[ 942    0   17    4    0   20   17    5   17   18]
 [   0 1107   56   15   23   17    9   38   54   10]
 [   2    2  809   26    6    2   10   18    9    2]
 [   2    2   28  887    3   84    0    8   32   15]
 [   1    1   16    2  872   19   21   20   27   72]
 [   7    1    0   14    5  624   20    0   42    1]
 [  15    5   42    9   10   22  872    1   15    1]
 [   2    2   21   21    2   13    0  877   12   76]
 [   7   15   39   21   13   69    9    3  743   13]
 [   2    0    4   11   48   22    0   58   23  801]]
number of errors: 1466
error rate is: 0.1466
```

The confusion matrix above is for the test data.

Performance Evaluation:

The error for the one-vs-all classifier has about 14.5% error rate while the one-vs-one classifier has about a 7.1% error rate for the test data. The one-vs-one classifier seems to out perform the one-vs-all classifier by twice the ammount.

Considering the test data: For the one-vs-one classifier, the number 1 seems to be the easiest to recognizes with 1122 correct guesses, while the most difficult to recognize is the number 5 with 788 correct guesses.

For the one-vs-all classifier, the easiset to recognize seems to be the number 1 as well with 1107 correct guesses, and the most difficult to guess is also the number 5 with 624 correct guesses.

It seems that numbers that look similar such as 5 and 8 have a lower amount of correct guesses, while numbers such as 1 or 0 have the best guesses. The one-vs-one classifier might outperform the one-vs-all classifier because it directly compares 2 numbers while the one-vs-all compares it to a bundle of numbers. The binary classification as such is more certain when it is one-to-one, since the appearance of multiple similar numbers may less affect the direct comparison

ECE174 MiniProject 1

Problem 2: Kmeans Clustering

Kwok Hung Ho A15151703

In [72]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
import random
from matplotlib import gridspec
import heapq
X = loadmat("mnist.mat")
data_train = X['trainX'] # Training set digits
data_label = X['trainY'] # Training set labels
data_test = X['testX'] # Training set digits
data_test_label = X['testY'] # Training set labels
inv   = np.linalg.inv
pinv  = np.linalg.pinv
solve = np.linalg.solve
rank  = np.linalg.matrix_rank
norm  = np.linalg.norm
det   = np.linalg.det
solve = np.linalg.solve
#ew_column1 = np.ones(60000)
#ata_train = np.insert(data_train, -1, new_column1, axis=1)
#ew_column2 = np.ones(10000)
#ata_test = np.insert(data_test, -1, new_column2, axis=1)
```

Functions are declared below:

Paticularly, grouprep stores the centroids while newcluster assigns new datapoints to the cluster. These 2 functions are looped and count as 1 iteration in main.

In [194]:

```
def euclideandist(x1, x2):
    #is this jclust?
    #divide by N (60,000)?
    return np.sqrt(np.sum((x1-x2)**2))

def kmeansinit(Nvectors, Kclusters):
    C = []
    for i in range(len(Nvectors)):
        ran = random.randint(1, Kclusters)
        C.append(ran)
    C = np.array(C)
    return C

def grouprep(data, C, K):
    Z = np.array([])
    for i in range(K):
        group = []
        for j in range(len(data)):
            if C[j] == i+1:
                group.append(data[j,:])
        group=np.array(group)
        centroid = np.mean(group, axis=0)
        Z = np.append(Z, centroid)
    Z = Z.reshape(K,784)
    return Z

def newcluster(data, Z, oldcluster, clusters):
    C = []
    jclust=[]
    distances = []
    for j in range(len(data)):
        distances = []
        for i in range(clusters):
            distances.append(euclideandist(data[j], Z[i]))
            if oldcluster[j]==i+1:
                jclust.append(euclideandist(data[j]/255,Z[i]/255)**2) #normalized
        minindex = np.argmin(distances)+1
        C.append(minindex)
    C = np.array(C)
    jclust = sum(jclust)/len(data)
    return C, jclust


#loop this P times
def main(data, K):
    initialC = kmeansinit(data, K)
    updatecentroid = grouprep(data, initialC, K)
    updatecluster, jclust = newcluster(data, updatecentroid, initialC, K)
    jclust=0
    jclustaxis=[]
    iterationaxis = []
    previousjclust=0
    for iterations in range(15):
        #print("iteration "+str(iterations) +": "+"jclust:" + str(jclust))
        updatecluster,jclust = newcluster(data, updatecentroid, updatecluster, K)
        updatecentroid = grouprep(data, updatecluster, K)
        print("iteration "+str(iterations) +": "+"jclust:" + str(jclust))
        jclustaxis.append(jclust)
        iterationaxis.append(iterations)
```

```
        if iterations > 0:
            previousjclust = jclustaxis[iterations - 1]

        if abs(jclust - previousjclust) < 0.1: #jclust convergence variable
            #print(jclust)
            #print(previousjclust)
            #print("break")
            break
        #print("")

    #plotting jclust
    jclustaxis = np.array(jclustaxis)
    iterationaxis = np.array(iterationaxis)
    #fig,ax = plt.subplots(1)
    #ax.plot(iterationaxis,jclustaxis)
    finalcentroid = updatecentroid
    finalcluster = updatecluster
    return finalcentroid, finalcluster, jclust, jclustaxis, iterationaxis

def plotjclust(jclustaxis, iterationaxis):
    fig,ax = plt.subplots(1)
    ax.plot(iterationaxis,jclustaxis)
    return
```

Running main function 30 times (P=30, K=20)

Below, we run main 30 times and save the outputs of the jclust changes each iteration, final jclust and final centroids each time. This allows us to find the argument of the maximum and minimum values for the final jclust in the paticular list and find the corresponding axes for plotting that specific "P" run. It also saves the final centroids aka group representatives so that we can reshape them into images later for inspection and evaluation of performance.

In [220]:

```
#Running Main 30 times (P=30, K=20)
finaljclust = []
jclustaxes = []
iterationaxes = []
finalcentroids = []
for P in range(30):
    print("P = " +str(P+1)+" :")
    finalcentroid, finalcluster, jclust, jclustaxis, iterationaxis= main(data_test, 20)
    finaljclust.append(jclust)
    jclustaxes.append(jclustaxis)
    iterationaxes.append(iterationaxis)
    finalcentroids.append(finalcentroid)
```

```
P = 1 :
iteration 0: jclust:51.417169987275564
iteration 1: jclust:41.792206239000826
iteration 2: jclust:37.73502875213554
iteration 3: jclust:36.50680774850369
iteration 4: jclust:35.928003166835985
iteration 5: jclust:35.527540020281386
iteration 6: jclust:35.2566509577559
iteration 7: jclust:35.10879296650794
iteration 8: jclust:35.03044768200095
P = 2 :
iteration 0: jclust:51.28625193439701
iteration 1: jclust:41.770989198188495
iteration 2: jclust:38.23498052297923
iteration 3: jclust:37.009187478016486
iteration 4: jclust:36.209891625499786
iteration 5: jclust:35.78297088059865
iteration 6: jclust:35.583088445539
iteration 7: jclust:35.45922793196302
iteration 8: jclust:35.379443460710775
P = 3 :
iteration 0: jclust:51.37875904225627
iteration 1: jclust:42.135337943304805
iteration 2: jclust:38.098891812539314
iteration 3: jclust:36.610441861976064
iteration 4: jclust:35.960277772754026
iteration 5: jclust:35.6669805898162
iteration 6: jclust:35.497603699882376
iteration 7: jclust:35.38120461389443
iteration 8: jclust:35.29784494537861
P = 4 :
iteration 0: jclust:51.42257117285596
iteration 1: jclust:42.35102122473415
iteration 2: jclust:38.042474329523856
iteration 3: jclust:36.72038038702851
iteration 4: jclust:35.91408078073306
iteration 5: jclust:35.513163812008465
iteration 6: jclust:35.39119287455569
iteration 7: jclust:35.33294551298098
P = 5 :
iteration 0: jclust:51.39959465933626
iteration 1: jclust:41.99019691600265
iteration 2: jclust:38.367962011978754
iteration 3: jclust:36.94456555591854
iteration 4: jclust:36.42539346532554
iteration 5: jclust:36.130877002351255
iteration 6: jclust:35.94145655327293
iteration 7: jclust:35.79348562457639
iteration 8: jclust:35.63148086875506
iteration 9: jclust:35.4813817843854
iteration 10: jclust:35.3769751752772
iteration 11: jclust:35.31340578240413
P = 6 :
iteration 0: jclust:51.384475076129554
iteration 1: jclust:42.3275382210409
iteration 2: jclust:38.16901058890516
iteration 3: jclust:36.861670047767134
iteration 4: jclust:36.353010874116535
iteration 5: jclust:35.983597203572934
iteration 6: jclust:35.74223850864477
iteration 7: jclust:35.60807841934213
```

```
iteration 8: jclust:35.49139797426626
iteration 9: jclust:35.376374134430314
iteration 10: jclust:35.318505940578945
P = 7 :
iteration 0: jclust:51.37556138031458
iteration 1: jclust:41.89933010237792
iteration 2: jclust:38.226096345103876
iteration 3: jclust:36.80602367434
iteration 4: jclust:36.15226836088441
iteration 5: jclust:35.850789346906
iteration 6: jclust:35.691609767957445
iteration 7: jclust:35.60300039559258
P = 8 :
iteration 0: jclust:51.25826871946447
iteration 1: jclust:41.39522376405319
iteration 2: jclust:37.71304130397227
iteration 3: jclust:36.34641140336919
iteration 4: jclust:35.7716536217025
iteration 5: jclust:35.51589626756477
iteration 6: jclust:35.35143697348721
iteration 7: jclust:35.21600855097068
iteration 8: jclust:35.11476200100175
iteration 9: jclust:35.06684936197966
P = 9 :
iteration 0: jclust:51.490860841985274
iteration 1: jclust:41.777109594876286
iteration 2: jclust:37.84856259791181
iteration 3: jclust:36.54235812892025
iteration 4: jclust:35.91455766794904
iteration 5: jclust:35.521436935743665
iteration 6: jclust:35.35232256667905
iteration 7: jclust:35.28436898409845
P = 10 :
iteration 0: jclust:51.46643026849033
iteration 1: jclust:42.32752365124982
iteration 2: jclust:38.188659527107546
iteration 3: jclust:36.982706697001156
iteration 4: jclust:36.36441776255212
iteration 5: jclust:35.95361898159076
iteration 6: jclust:35.62330556909936
iteration 7: jclust:35.40145450230205
iteration 8: jclust:35.23624444125694
iteration 9: jclust:35.10465282824978
iteration 10: jclust:35.027335034589036
P = 11 :
iteration 0: jclust:51.370049913725026
iteration 1: jclust:42.06007080847333
iteration 2: jclust:37.83235137123549
iteration 3: jclust:36.351590849775796
iteration 4: jclust:35.764109044244236
iteration 5: jclust:35.51954535893773
iteration 6: jclust:35.36706488069872
iteration 7: jclust:35.267131544178774
P = 12 :
iteration 0: jclust:51.31206376216039
iteration 1: jclust:42.473040208951225
iteration 2: jclust:38.44722719839591
iteration 3: jclust:36.874881142607194
iteration 4: jclust:36.01420470574709
iteration 5: jclust:35.554851194962666
iteration 6: jclust:35.314576303149494
```

```
iteration 7: jclust:35.164525102624296
iteration 8: jclust:35.043620168901484
iteration 9: jclust:34.95591600262091
P = 13 :
iteration 0: jclust:51.35157163643282
iteration 1: jclust:41.95525441504384
iteration 2: jclust:37.90118705331554
iteration 3: jclust:36.738937935234766
iteration 4: jclust:36.25007138247478
iteration 5: jclust:36.01167932253301
iteration 6: jclust:35.84110869340167
iteration 7: jclust:35.69922399406949
iteration 8: jclust:35.56553656830903
iteration 9: jclust:35.434335866906714
iteration 10: jclust:35.32429704500379
iteration 11: jclust:35.22504713654144
P = 14 :
iteration 0: jclust:51.15497254731501
iteration 1: jclust:41.7633991236349
iteration 2: jclust:37.59223909422538
iteration 3: jclust:36.43572605179577
iteration 4: jclust:35.97794291448421
iteration 5: jclust:35.70487248486052
iteration 6: jclust:35.49927928568614
iteration 7: jclust:35.38912524810343
iteration 8: jclust:35.336093772448834
P = 15 :
iteration 0: jclust:51.35300426948481
iteration 1: jclust:42.48315556497168
iteration 2: jclust:38.17398749622213
iteration 3: jclust:36.88876585160899
iteration 4: jclust:36.29909096300787
iteration 5: jclust:35.90560270090849
iteration 6: jclust:35.63007440357949
iteration 7: jclust:35.44084930785666
iteration 8: jclust:35.34558742916557
P = 16 :
iteration 0: jclust:51.380556295243956
iteration 1: jclust:42.06175440552123
iteration 2: jclust:38.20602603657297
iteration 3: jclust:36.94413591591522
iteration 4: jclust:36.316987856580454
iteration 5: jclust:35.98631798885238
iteration 6: jclust:35.79982136512306
iteration 7: jclust:35.67883567627001
iteration 8: jclust:35.59677658632893
P = 17 :
iteration 0: jclust:51.29990819733906
iteration 1: jclust:41.667735445091026
iteration 2: jclust:37.857738870352335
iteration 3: jclust:36.791235518667186
iteration 4: jclust:36.20238136221321
iteration 5: jclust:35.74677167492673
iteration 6: jclust:35.470554699855626
iteration 7: jclust:35.35366572065896
iteration 8: jclust:35.286783772778
P = 18 :
iteration 0: jclust:51.433394178525695
iteration 1: jclust:42.05375470153241
iteration 2: jclust:37.763137789000595
iteration 3: jclust:36.781392741844044
```

```
iteration 4: jclust:36.25552232949623
iteration 5: jclust:35.87333878868174
iteration 6: jclust:35.645250743984775
iteration 7: jclust:35.517828161667985
iteration 8: jclust:35.42925625332147
P = 19 :
iteration 0: jclust:51.368756442427525
iteration 1: jclust:41.73064957510201
iteration 2: jclust:38.06229686333489
iteration 3: jclust:36.843507843587815
iteration 4: jclust:36.06052998909289
iteration 5: jclust:35.62020165234452
iteration 6: jclust:35.428734822079385
iteration 7: jclust:35.32171291575465
iteration 8: jclust:35.243931748682044
P = 20 :
iteration 0: jclust:51.377421601920155
iteration 1: jclust:42.57318853909128
iteration 2: jclust:38.47804428332968
iteration 3: jclust:36.71256619557722
iteration 4: jclust:36.11926581168314
iteration 5: jclust:35.81991939533757
iteration 6: jclust:35.63580701847136
iteration 7: jclust:35.50468963641348
iteration 8: jclust:35.415444005858156
P = 21 :
iteration 0: jclust:51.40397047133739
iteration 1: jclust:42.174593193825494
iteration 2: jclust:38.53117170086518
iteration 3: jclust:37.12304136908873
iteration 4: jclust:36.24422076979178
iteration 5: jclust:35.80158266313772
iteration 6: jclust:35.63023987992392
iteration 7: jclust:35.534698532174374
P = 22 :
iteration 0: jclust:51.21623407454693
iteration 1: jclust:41.481700779525816
iteration 2: jclust:37.79984230765536
iteration 3: jclust:36.6639893210398
iteration 4: jclust:36.086872236393354
iteration 5: jclust:35.787594284040935
iteration 6: jclust:35.608868178519536
iteration 7: jclust:35.50697477614305
iteration 8: jclust:35.45586097465148
P = 23 :
iteration 0: jclust:51.27356673510241
iteration 1: jclust:42.258163034684735
iteration 2: jclust:38.295321197830894
iteration 3: jclust:37.23049111407151
iteration 4: jclust:36.58721935984275
iteration 5: jclust:36.05785897905593
iteration 6: jclust:35.73805189551717
iteration 7: jclust:35.51763004606976
iteration 8: jclust:35.3408076013177
iteration 9: jclust:35.24816566541368
P = 24 :
iteration 0: jclust:51.21088286519332
iteration 1: jclust:41.88045164764707
iteration 2: jclust:37.55673377218195
iteration 3: jclust:36.30284705424336
iteration 4: jclust:35.875244475727214
```

```
iteration 5: jclust:35.63754427347919
iteration 6: jclust:35.430261306800965
iteration 7: jclust:35.27040761165771
iteration 8: jclust:35.170033262873474
iteration 9: jclust:35.103056316765766
P = 25 :
iteration 0: jclust:51.43732121512332
iteration 1: jclust:41.557890305430625
iteration 2: jclust:37.665959428932524
iteration 3: jclust:36.42138623817835
iteration 4: jclust:35.71493494427714
iteration 5: jclust:35.397639043906146
iteration 6: jclust:35.24156961712955
iteration 7: jclust:35.153775890281494
P = 26 :
iteration 0: jclust:51.42371072187255
iteration 1: jclust:42.20003306542921
iteration 2: jclust:37.91523946722659
iteration 3: jclust:36.519167686719534
iteration 4: jclust:35.91675278621923
iteration 5: jclust:35.597383662024654
iteration 6: jclust:35.38247140221904
iteration 7: jclust:35.235132049786124
iteration 8: jclust:35.15580224656371
P = 27 :
iteration 0: jclust:51.26920442832596
iteration 1: jclust:42.17780564822998
iteration 2: jclust:38.27179767941118
iteration 3: jclust:36.830036433469665
iteration 4: jclust:36.17645841727837
iteration 5: jclust:35.821270669187804
iteration 6: jclust:35.59414402429735
iteration 7: jclust:35.40795655798516
iteration 8: jclust:35.27218754774426
iteration 9: jclust:35.203814218656916
P = 28 :
iteration 0: jclust:51.43860112444985
iteration 1: jclust:42.406392357321096
iteration 2: jclust:38.36219466637304
iteration 3: jclust:37.06575818631273
iteration 4: jclust:36.305132085384194
iteration 5: jclust:35.892568710132345
iteration 6: jclust:35.63306289010334
iteration 7: jclust:35.473205973161186
iteration 8: jclust:35.3779855633934
P = 29 :
iteration 0: jclust:51.41450986095018
iteration 1: jclust:42.771644974076686
iteration 2: jclust:38.444584923176464
iteration 3: jclust:36.47240140198014
iteration 4: jclust:35.75056683614242
iteration 5: jclust:35.46677649540309
iteration 6: jclust:35.33068167860463
iteration 7: jclust:35.254355757089904
P = 30 :
iteration 0: jclust:51.346431945874706
iteration 1: jclust:41.24888649743889
iteration 2: jclust:38.42157652502716
iteration 3: jclust:37.24580063455938
iteration 4: jclust:36.46755353765322
iteration 5: jclust:36.012017781878626
```

```
iteration 6: jclust:35.63797287590465
iteration 7: jclust:35.37209476748347
iteration 8: jclust:35.26950395491077
iteration 9: jclust:35.18060989386997
```

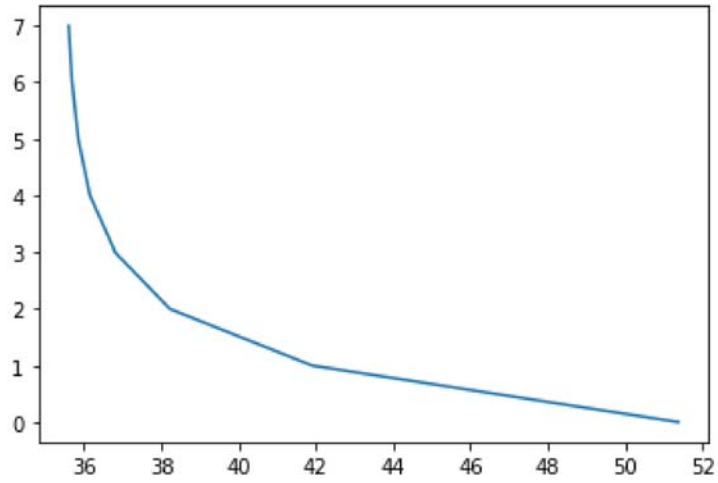
Results of P=30, K=20:

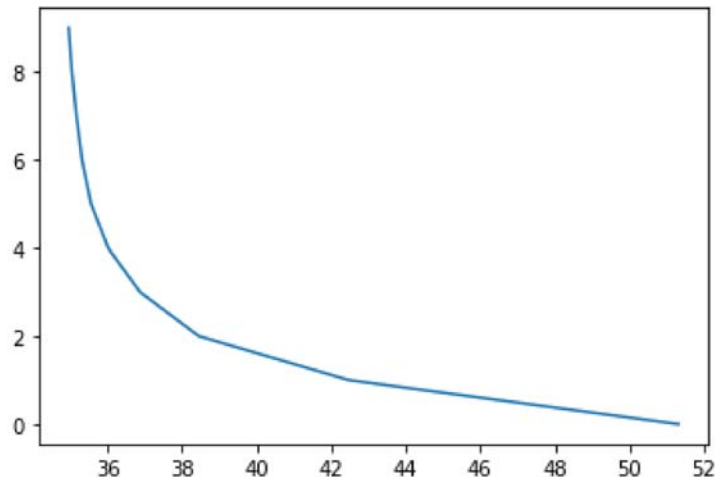
As computed from above, below, for the maximum and minimum JClust runs which are P=7 and P=12 respectively, corresponding to the index6 and index11 respectively, has jclust values 35.612 and 34.95 respectively. The plots are as shown respectively. The top plot corresponds to the max run and the bottom plot corresponds to the min run. The X axis for both is the Jclust value and the Y axis is the iteration. The plots move from right to left over time. The Jclust values are also normalized to 1 order of magnitude because in newcluster, where I compute it, I normalized the data and the group representatives to be between 0 and 1.

In [221]:

```
# Retrieving Maxima and Minima
npfinaljclust=np.array(finaljclust)

maxjclust = max(finaljclust)
maxindex=np.argmax(npfinaljclust)
plotjclust(iterationaxes[maxindex],jclustaxes[maxindex])##
maxcentroid = finalcentroids[maxindex]

minjclust =min(finaljclust)
minindex=np.argmin(npfinaljclust)
plotjclust(iterationaxes[minindex],jclustaxes[minindex])##
mincentroid = finalcentroids[minindex]

print(maxindex)
print(minindex)
```

6
11



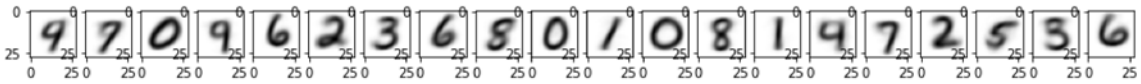


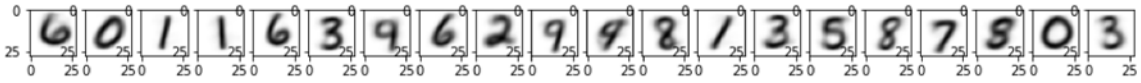
Below are the final group representatives for the max and min runs after their final iteration. Converging with a Jclust of about 35, the pictures below resemble definite handwritten numbers with the exception of a few. The top 20 images are the 20 K group representatives aka centroids for the maximum jclust returned out of 30 runs. The bottom 20 images are the 20 K group representatives aka centroids for the maximum jclust returned out of 30 runs.

Group representatives plotted:

In [222]:

```
#Visualize K group representatives as images (centroids)
K=20
maxcentroid = finalcentroids[maxindex]
mincentroid = finalcentroids[minindex]
fig = plt.figure()
fig.set_figheight(15)
fig.set_figwidth(15)
for i in range(K):
    fig.add_subplot(1,20, 1+i)
    plt.imshow(maxcentroid[i].reshape(28,28),cmap='binary')
    fig.add_subplot(2,20, 1+i)
    plt.imshow(mincentroid[i].reshape(28,28),cmap='binary')
```





Discussion of Results:

The majority of the pictures are distinguishable, the exceptions include 9 and 4 being similar, and 5 and 8 being similar. Though not as clear as the training data, these points are averages from the clusters, having similar features being grouped. The result is not bad and the algorithm succeeds.

Finding closest 10 data points:

Below is the code that creates the numpy array npmin10, which stores the distances of every data point to every centroid.

In [223]:

```
K = 20
min10 = []
finalmin10=[]
for i in range(len(data_test)):
    distances = []
    for j in range(K):
        distances.append(euclideandist(data_test[i],mincentroid[j]))
    min10.append(distances)#10,000 nparrays of20
npmin10 = np.array(min10)

min10 = []
finalmin10=[]
for i in range(len(data_test)):
    distances = []
    for j in range(K):
        distances.append(euclideandist(data_test[i],maxcentroid[j]))
    min10.append(distances)#10,000 nparrays of20
npmax10 = np.array(min10)
```

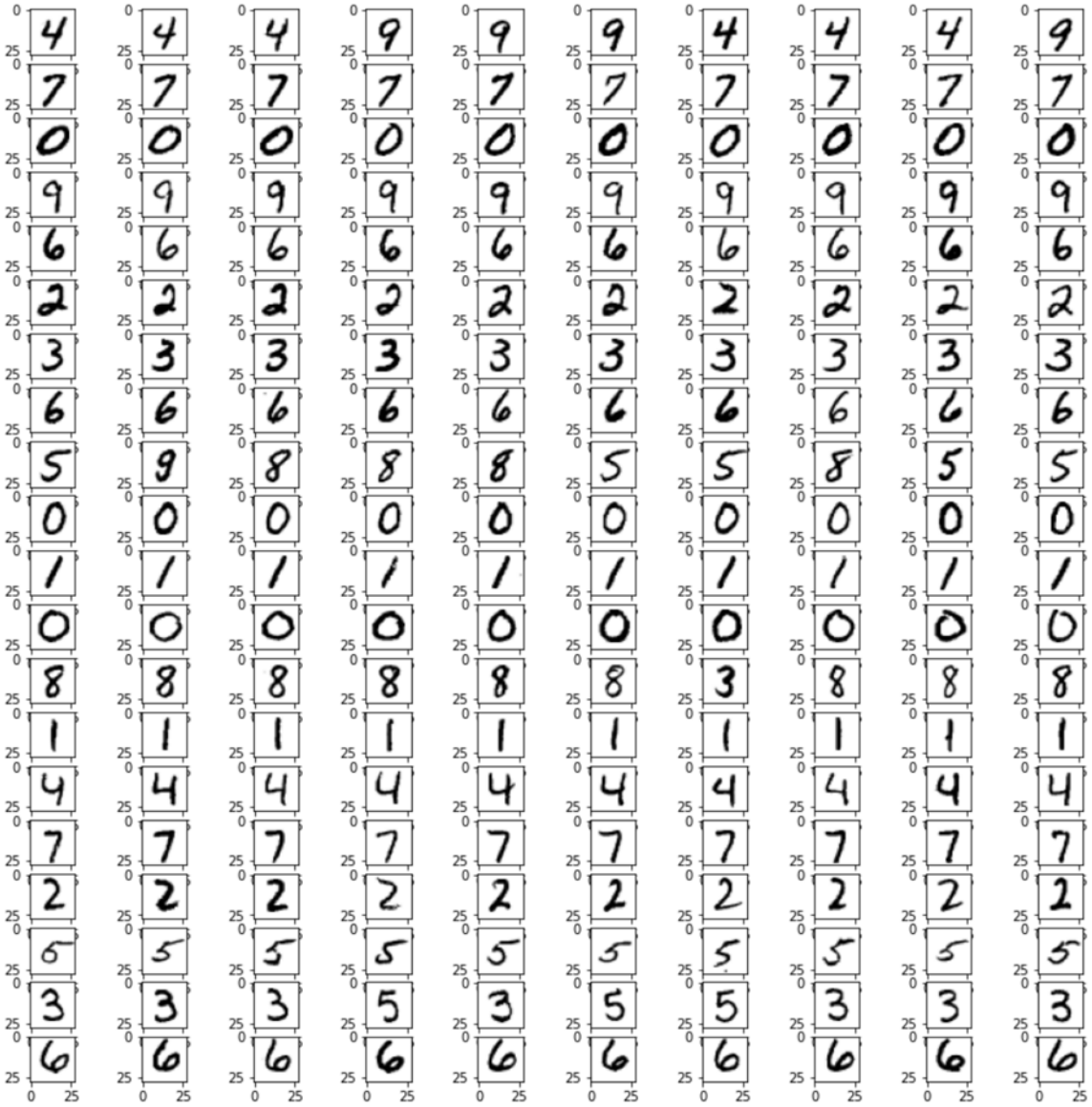
Plotting the images, eyeballing digits (P=30, K=20)

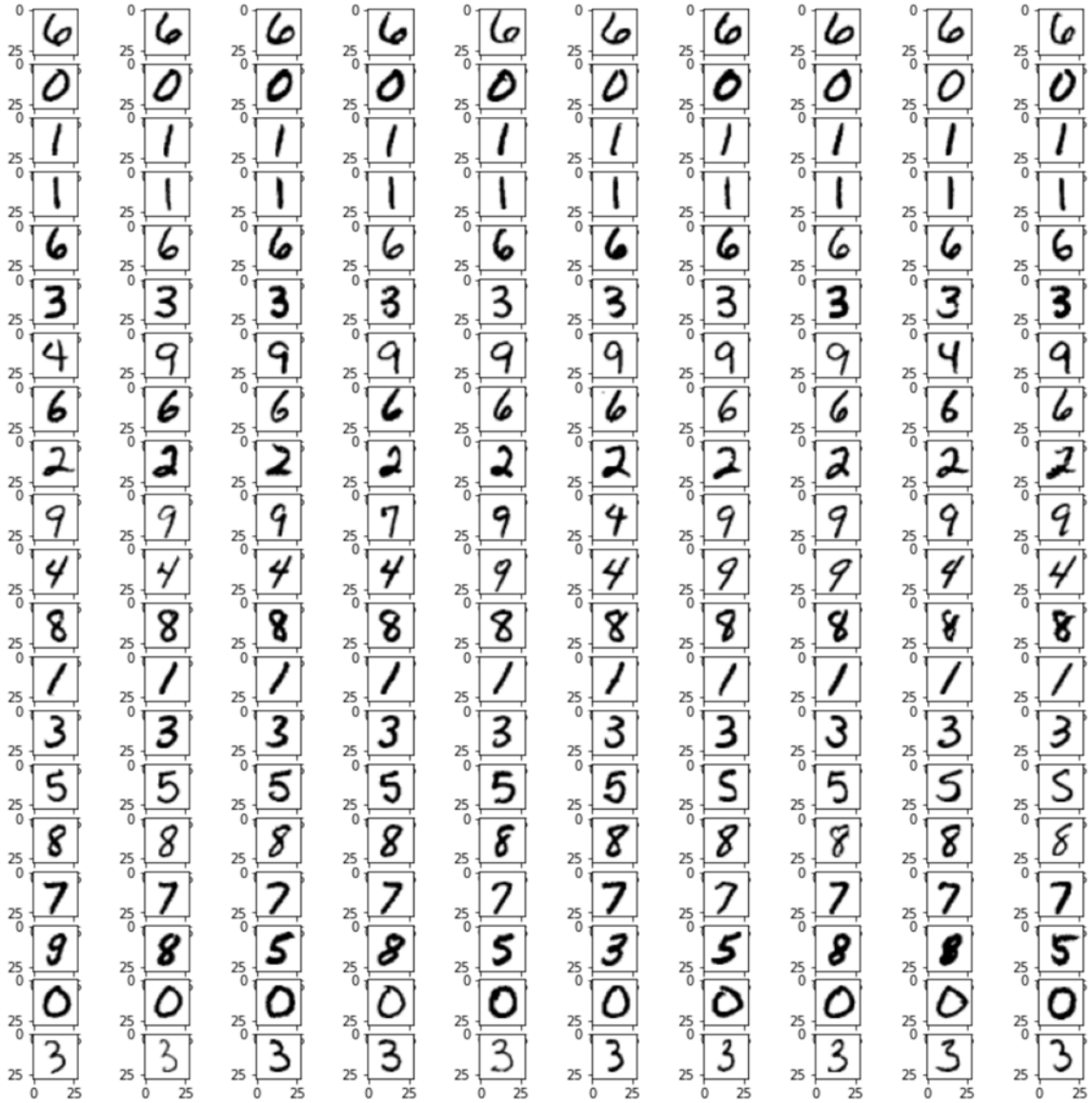
Using argsort on the negation of npmin10, I return the indices of the 10 smallest distances for each cluster. (Since argsort returns the max value I had to negate the matrix). I then plot the corresponding datapoints in a 20X10 figure. Each row indicates the Kth cluster and should have similar features via Kmeans algorithm.

In [228]:

```
temp=[]
fig, axs = plt.subplots(20, 10, figsize=(15,15))
for i in range(20):
    a = (-npmin10[:,i]).argsort()[-10:][::-1]
    temp.append(a) #gives index of 10 closest pictures each cluster
    print(temp[i])
for i in range(20):
    for j in range(10):
         axs[i,j].imshow(data_test[temp[i][j],:].reshape(28,28),cmap='binary')

print("min-max")

temp=[]
fig, axs = plt.subplots(20, 10, figsize=(15,15))
for i in range(20):
    a = (-npmax10[:,i]).argsort()[-10:][::-1]
    temp.append(a) #gives index of 10 closest pictures each cluster
    print(temp[i])
for i in range(20):
    for j in range(10):
         axs[i,j].imshow(data_test[temp[i][j],:].reshape(28,28),cmap='binary')
```

```
[8879 5362 1111 1183 1745 6318 7833 6974 2615 6520]
[6306 5110 7124 5107 7544 7238 8825 7059 1848 5077]
[5124 8978 5069 7601 8343 6766 8137 8764 5139 8772]
[1165 8623 3041 1668 2443 2792 2692  773 7322  481]
[5276 9312 2458 3395 6252 6933 2759 1373 8779 5354]
[8746 4296 6193 2981 6217 2662 3142 7012  208 5723]
[9365 6982 8701 8070 5346 8876 8910 1205 5508 8811]
[9149 3382 8814 5378 2026 6707 7358 6639 6943 9148]
[9298  448 6442 7060 3224 1693 1041 5059 7274  129]
[5721 3959 1294 1739 3481 3310 1261  148 5592 7703]
[5461 6276 8432 3211 7091 9802 7165  202 5113 5316]
[4033 1708 8468 9230 1333 9459   71 8548 4611 7693]
[9420 3374 5875 5864 2856 9330 2549  621 3002 8618]
[4931 1350 3919 4774  342 3648  675 5524 1994  663]
[4192  843 1815 7621   49 9276 3684 1652 5202 3278]
[6586 2567 9806 2275 9288 9340 3993 1799 7650 6856]
[2092 2481 4456 1056 2157 2296  306 1927 4354 8069]
[8275 9675 9616 2241 5658 8982 6530 9671 9606 8232]
[1605 5606 6863 3312 3944 3917 1144 2083 4770 3715]
[8668 6301 5884  463 6387 9373 6354 6325 5609 1921]
min-max
[8668 9373 6387 5161 3822 5481 6301 1921 5441  472]
[7601 6454 7385 8772 6786 7086 2087 8731  794 7052]
[ 672 9464 6407 2880 8658 7722 2357 8740 7290 1633]
[ 735  427 4603 4085 9443 3454  239 3919 5524 5896]
[5276 2458 6933 9312 3395 8779 3362 1373 6252 4908]
[8777 4818 7936 2853 2076  855 3715 7312 5766   68]
[3282  105 3485 3759 4245  496 2323 4616 9406 9472]
[9149 3382 4209 6707 2026 8814 6639 4132 3908 7546]
[ 208 6193 3142 4296 7281   82 6534 6217 3223 9623]
[4901  471 4184 2808 4843 1270 9362 5414 7707 1906]
[5101 1580 5047 6951 4016 8917 5247 9767 9922 8993]
[9418 5864 2146  793 6838 4372  872 9173 2739 2211]
[9485 7165 4262 6502 7442 6468 9030 5316 6789 9969]
[9365 8701 4581 8461 1442 5346 6875 8533 8864 6930]
[3917 1144  395 1510 7659 5528  240 3312 2077  397]
[3588 4850 7060 6275 9824 5489  961 6612 9420 9854]
[5795 5808  223 7059 5372 7141 8234 7124 7362 5277]
[ 448 7765 8676 7378 7324 6184 7397 7297 8105 1376]
[1333 4542   71 7699 3179 5930 4611 9367 8595 4070]
[ 205 4630 6863 2083 1513 6882  683   76 1531 9275]
```





Generating a table:

Through eyeballing the data and comparing to the second row corresponding the minium run centroids from above. I have generated a 2X20 table below to indicate the correctly and incorectly classified closest 10 data points. The Top row signifies the correct amount of classifications and the bottom signifies the amount of wrongly classified. We can see that the worst performing group was group index7, index11 and index13 with 4 wrong classifications in my opinion. This makes sense as the corresponding centroids as seen fomr above are the most blurry group representatives.

In [231]:

```
min_correct_table = np.zeros(20)
min_correct_table[0] = 6
min_correct_table[1] = 10
min_correct_table[2] = 10
min_correct_table[3] = 10
min_correct_table[4] = 10
min_correct_table[5] = 10
min_correct_table[6] = 10
min_correct_table[7] = 10
min_correct_table[8] = 5
min_correct_table[9] = 10
min_correct_table[10] = 10
min_correct_table[11] = 10
min_correct_table[12] = 9
min_correct_table[13] = 10
min_correct_table[14] = 10
min_correct_table[15] = 10
min_correct_table[16] = 10
min_correct_table[17] = 10
min_correct_table[18] = 7
min_correct_table[19] = 10
min_misclassified_table = 10 - min_correct_table
print("min:")
min_correct_table = min_correct_table.astype(int)
min_misclassified_table=min_misclassified_table.astype(int)
print(min_correct_table)
print(min_misclassified_table)
max_correct_table = np.zeros(20)
max_correct_table[0] = 10
max_correct_table[1] = 10
max_correct_table[2] = 10
max_correct_table[3] = 10
max_correct_table[4] = 10
max_correct_table[5] = 10
max_correct_table[6] = 9
max_correct_table[7] = 10
max_correct_table[8] = 10
max_correct_table[9] = 8
max_correct_table[10] = 7
max_correct_table[11] = 10
max_correct_table[12] = 10
max_correct_table[13] = 10
max_correct_table[14] = 10
max_correct_table[15] = 10
max_correct_table[16] = 10
max_correct_table[17] = 4
max_correct_table[18] = 10
max_correct_table[19] = 10
max_misclassified_table = 10 - max_correct_table
print("max:")
max_correct_table = max_correct_table.astype(int)
max_misclassified_table=max_misclassified_table.astype(int)
print(max_correct_table)
print(max_misclassified_table)
```

```
min:
[ 6 10 10 10 10 10 10  5 10 10 10  9 10 10 10 10  7 10]
[4 0 0 0 0 0 0 0 5 0 0 0 1 0 0 0 0 0 3 0]
max:
[10 10 10 10 10 10  9 10 10  8  7 10 10 10 10 10 10  4 10 10]
[0 0 0 0 0 0 1 0 0 2 3 0 0 0 0 0 0 6 0 0]
```

Repeating for K =10 and P =20

In [195]:

```
#Running Main 30 times (P=20, K=10)
finaljclust = []
jclustaxes = []
iterationaxes = []
finalcentroids = []
for P in range(20):
    print("P = " +str(P+1)+" :")
    finalcentroid, finalcluster, jclust, jclustaxis, iterationaxis= main(data_test, 10)
    finaljclust.append(jclust)
    jclustaxes.append(jclustaxis)
    iterationaxes.append(iterationaxis)
    finalcentroids.append(finalcentroid)
```

```
P = 1 :
iteration 0: jclust:52.080202893565
iteration 1: jclust:45.17110507194459
iteration 2: jclust:41.89492615874467
iteration 3: jclust:41.02120712069119
iteration 4: jclust:40.64055445483258
iteration 5: jclust:40.37949209427002
iteration 6: jclust:40.153543061948035
iteration 7: jclust:39.926822576441026
iteration 8: jclust:39.72897616704789
iteration 9: jclust:39.52686884391933
iteration 10: jclust:39.333116548623124
iteration 11: jclust:39.17996877617728
iteration 12: jclust:39.08084161573949
P = 2 :
iteration 0: jclust:51.95517767051623
iteration 1: jclust:44.17513137957651
iteration 2: jclust:40.99422567074636
iteration 3: jclust:39.80438990157642
iteration 4: jclust:39.39086765574237
iteration 5: jclust:39.241885021621506
iteration 6: jclust:39.167472586838315
P = 3 :
iteration 0: jclust:51.9460702217003
iteration 1: jclust:45.35276871791553
iteration 2: jclust:41.70945255796379
iteration 3: jclust:40.552161627443155
iteration 4: jclust:39.901258390863994
iteration 5: jclust:39.47257184112591
iteration 6: jclust:39.26791579976332
iteration 7: jclust:39.20233692518337
P = 4 :
iteration 0: jclust:51.9067657501172
iteration 1: jclust:44.991170944353684
iteration 2: jclust:41.9726069189989
iteration 3: jclust:40.59730658228963
iteration 4: jclust:39.95595941289803
iteration 5: jclust:39.62674939291852
iteration 6: jclust:39.449153476399225
iteration 7: jclust:39.31598604883275
iteration 8: jclust:39.218951993047966
P = 5 :
iteration 0: jclust:51.945249305305445
iteration 1: jclust:45.25056076361406
iteration 2: jclust:41.87664768127953
iteration 3: jclust:40.768678163116604
iteration 4: jclust:40.087681714612415
iteration 5: jclust:39.55100940784165
iteration 6: jclust:39.29171398867988
iteration 7: jclust:39.16554468962497
iteration 8: jclust:39.073629026512485
P = 6 :
iteration 0: jclust:51.99977680103876
iteration 1: jclust:44.280271000774015
iteration 2: jclust:40.82052925805945
iteration 3: jclust:39.902768376714675
iteration 4: jclust:39.54758071265877
iteration 5: jclust:39.40562354509894
iteration 6: jclust:39.31205467011672
P = 7 :
iteration 0: jclust:51.80097798172185
```

```
iteration 1: jclust:44.61658447049802
iteration 2: jclust:42.035669927221704
iteration 3: jclust:40.84297714813763
iteration 4: jclust:40.037600745983084
iteration 5: jclust:39.648023030140266
iteration 6: jclust:39.499698479336445
iteration 7: jclust:39.42682672207578
P = 8 :
iteration 0: jclust:51.82716978126271
iteration 1: jclust:44.17307516414079
iteration 2: jclust:41.55495047651885
iteration 3: jclust:40.742919785805654
iteration 4: jclust:40.29455274903097
iteration 5: jclust:39.94537603968559
iteration 6: jclust:39.648724494966324
iteration 7: jclust:39.41747058593639
iteration 8: jclust:39.26841505698931
iteration 9: jclust:39.15938427912755
iteration 10: jclust:39.08943176419166
P = 9 :
iteration 0: jclust:51.9595985025685
iteration 1: jclust:44.36429675830719
iteration 2: jclust:41.449598878860435
iteration 3: jclust:40.77369821179288
iteration 4: jclust:40.447543237310256
iteration 5: jclust:40.264709321908356
iteration 6: jclust:40.13224154494293
iteration 7: jclust:40.010793037276194
iteration 8: jclust:39.86881271662978
iteration 9: jclust:39.68527108161975
iteration 10: jclust:39.459720388679685
iteration 11: jclust:39.281715265013226
iteration 12: jclust:39.21858053709965
P = 10 :
iteration 0: jclust:51.894471585936905
iteration 1: jclust:44.30480118905593
iteration 2: jclust:41.507898326146304
iteration 3: jclust:40.82123353316584
iteration 4: jclust:40.34062206589721
iteration 5: jclust:39.89027591362168
iteration 6: jclust:39.62080112148663
iteration 7: jclust:39.49316784146938
iteration 8: jclust:39.407480332663106
P = 11 :
iteration 0: jclust:51.99566341634541
iteration 1: jclust:43.78531843285428
iteration 2: jclust:40.82251454825056
iteration 3: jclust:39.82128918586689
iteration 4: jclust:39.51916221025781
iteration 5: jclust:39.411578043170685
iteration 6: jclust:39.33702903022912
P = 12 :
iteration 0: jclust:51.975979900494046
iteration 1: jclust:44.5685735334259
iteration 2: jclust:41.90055346724299
iteration 3: jclust:40.820605725691486
iteration 4: jclust:40.04031890000206
iteration 5: jclust:39.66670597557798
iteration 6: jclust:39.5124351791355
iteration 7: jclust:39.440439942579054
P = 13 :
```

```
iteration 0: jclust:51.86212777198661
iteration 1: jclust:44.320582915237765
iteration 2: jclust:41.42657621022896
iteration 3: jclust:40.34312650847605
iteration 4: jclust:39.75205744398424
iteration 5: jclust:39.432536352655006
iteration 6: jclust:39.29004602028914
iteration 7: jclust:39.21284355444531
P = 14 :
iteration 0: jclust:51.9899214762612
iteration 1: jclust:43.714156958168466
iteration 2: jclust:40.53781445460933
iteration 3: jclust:39.739443100868094
iteration 4: jclust:39.53566892296463
iteration 5: jclust:39.45422006928561
P = 15 :
iteration 0: jclust:51.89220691527469
iteration 1: jclust:43.67255768064417
iteration 2: jclust:40.776387874899726
iteration 3: jclust:39.79129462957146
iteration 4: jclust:39.46256325509221
iteration 5: jclust:39.35539965060324
iteration 6: jclust:39.30054066089049
P = 16 :
iteration 0: jclust:51.957211816590096
iteration 1: jclust:44.33680432751941
iteration 2: jclust:41.37929389769963
iteration 3: jclust:40.41735085318562
iteration 4: jclust:39.9646305347065
iteration 5: jclust:39.704946624402304
iteration 6: jclust:39.54028731906168
iteration 7: jclust:39.44073798354922
P = 17 :
iteration 0: jclust:52.00050228226282
iteration 1: jclust:44.579444579770666
iteration 2: jclust:41.514599922340835
iteration 3: jclust:40.15772171778104
iteration 4: jclust:39.583859768804544
iteration 5: jclust:39.363296304695574
iteration 6: jclust:39.23145723789883
iteration 7: jclust:39.160710143737205
P = 18 :
iteration 0: jclust:52.13166397753633
iteration 1: jclust:45.29092458762913
iteration 2: jclust:41.67977597006891
iteration 3: jclust:40.830125624198175
iteration 4: jclust:40.5398909424243
iteration 5: jclust:40.33765066881269
iteration 6: jclust:40.04145910371419
iteration 7: jclust:39.63777232030441
iteration 8: jclust:39.37989269262209
iteration 9: jclust:39.26474261677271
iteration 10: jclust:39.21847629940141
P = 19 :
iteration 0: jclust:51.9091334842061
iteration 1: jclust:45.587079428217706
iteration 2: jclust:42.25841577376728
iteration 3: jclust:40.743708203225026
iteration 4: jclust:39.944704725730766
iteration 5: jclust:39.54878871411438
iteration 6: jclust:39.40856613094562
```

```
iteration 7: jclust:39.3236745206172
P = 20 :
iteration 0: jclust:51.91670493429543
iteration 1: jclust:45.317275780030066
iteration 2: jclust:42.255111539040286
iteration 3: jclust:40.82855484491407
iteration 4: jclust:40.19627285250412
iteration 5: jclust:39.84481471353523
iteration 6: jclust:39.646780356251114
iteration 7: jclust:39.50405915332793
iteration 8: jclust:39.397241912148345
iteration 9: jclust:39.31246241564619
```

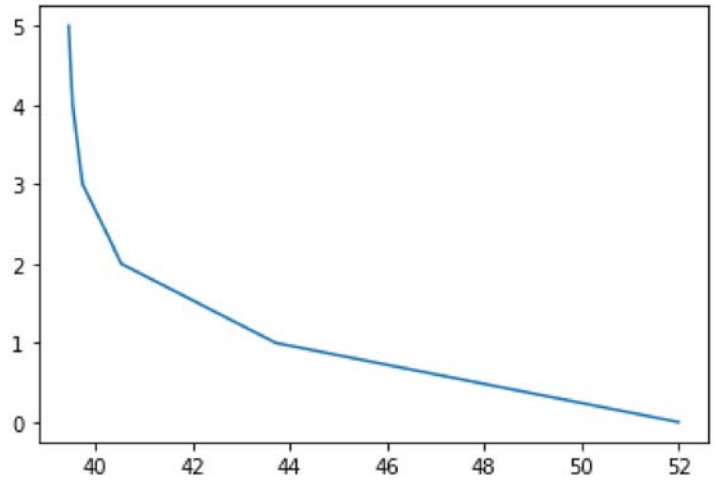
Results of P=20, K=10:

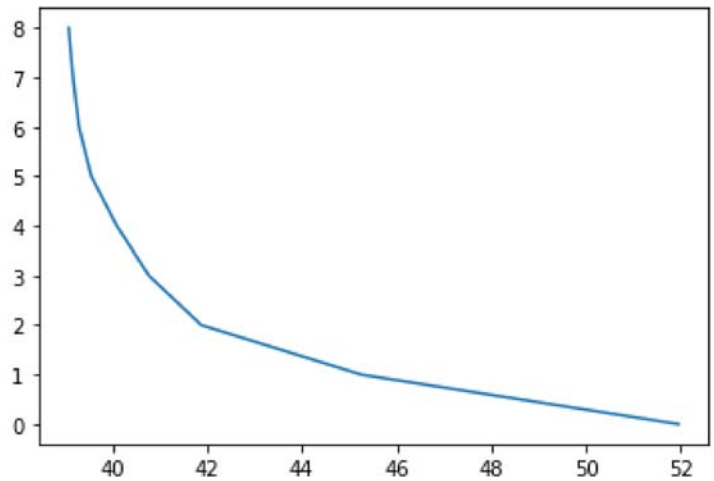
Max run with P = 14 has final jclust of 39.45 and min run witj P = 5 has final jclust of 39.07. The index thus is respectively 13 and 4 as printed below. The plots are also shown for the max and min respectively from top to bottom.

In [196]:

```
# Retrieving Maxima and Minima
npfinaljclust=np.array(finaljclust)

maxjclust = max(finaljclust)
maxindex=np.argmax(npfinaljclust)
plotjclust(iterationaxes[maxindex],jclustaxes[maxindex])##
maxcentroid = finalcentroids[maxindex]

minjclust =min(finaljclust)
minindex=np.argmin(npfinaljclust)
plotjclust(iterationaxes[minindex],jclustaxes[minindex])##
mincentroid = finalcentroids[minindex]

print(maxindex)
print(minindex)
```

```
13
4
```

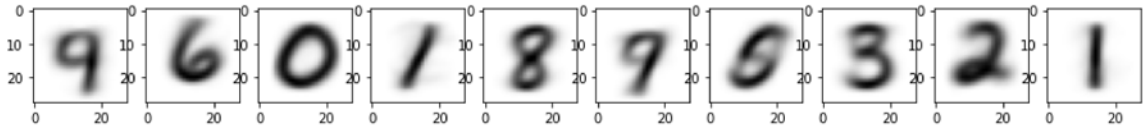


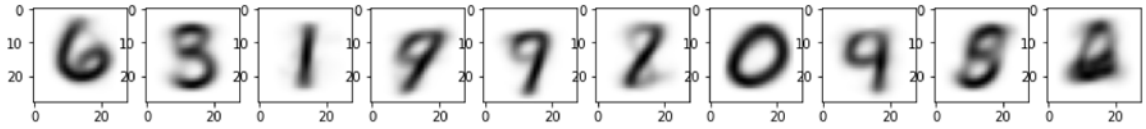


Below are the final group representatives for the max and min jclust runs respectively. The top row is max and bottom row is min.

In [197]:

```
K=10
maxcentroid = finalcentroids[maxindex]
mincentroid = finalcentroids[minindex]
fig = plt.figure()
fig.set_figheight(15)
fig.set_figwidth(15)
for i in range(K):
    fig.add_subplot(1,10, 1+i)
    plt.imshow(maxcentroid[i].reshape(28,28),cmap='binary')
    fig.add_subplot(2,10, 1+i)
    plt.imshow(mincentroid[i].reshape(28,28),cmap='binary')
```





Discussion of Results:

The majority of the centroids are quite distinguishable. With only 1 or 2 groups being blurry. This can have an affect on the classification accuracy later.

Finding closest 10 data points:

Below is the code that creates the numpy array npmin10, which stores the distances of every data point to every centroid.

In [200]:

```
K = 10
min10 = []
for i in range(len(data_test)):
    distances = []
    for j in range(K):
        distances.append(euclideandist(data_test[i],maxcentroid[j]))
    min10.append(distances)#10,000 nparrays of20
npmax10 = np.array(min10)

min10 = []
for o in range(len(data_test)):
    distances = []
    for f in range(K):
        distances.append(euclideandist(data_test[o],mincentroid[f]))
    min10.append(distances)#10,000 nparrays of20
npmin10 = np.array(min10)
```

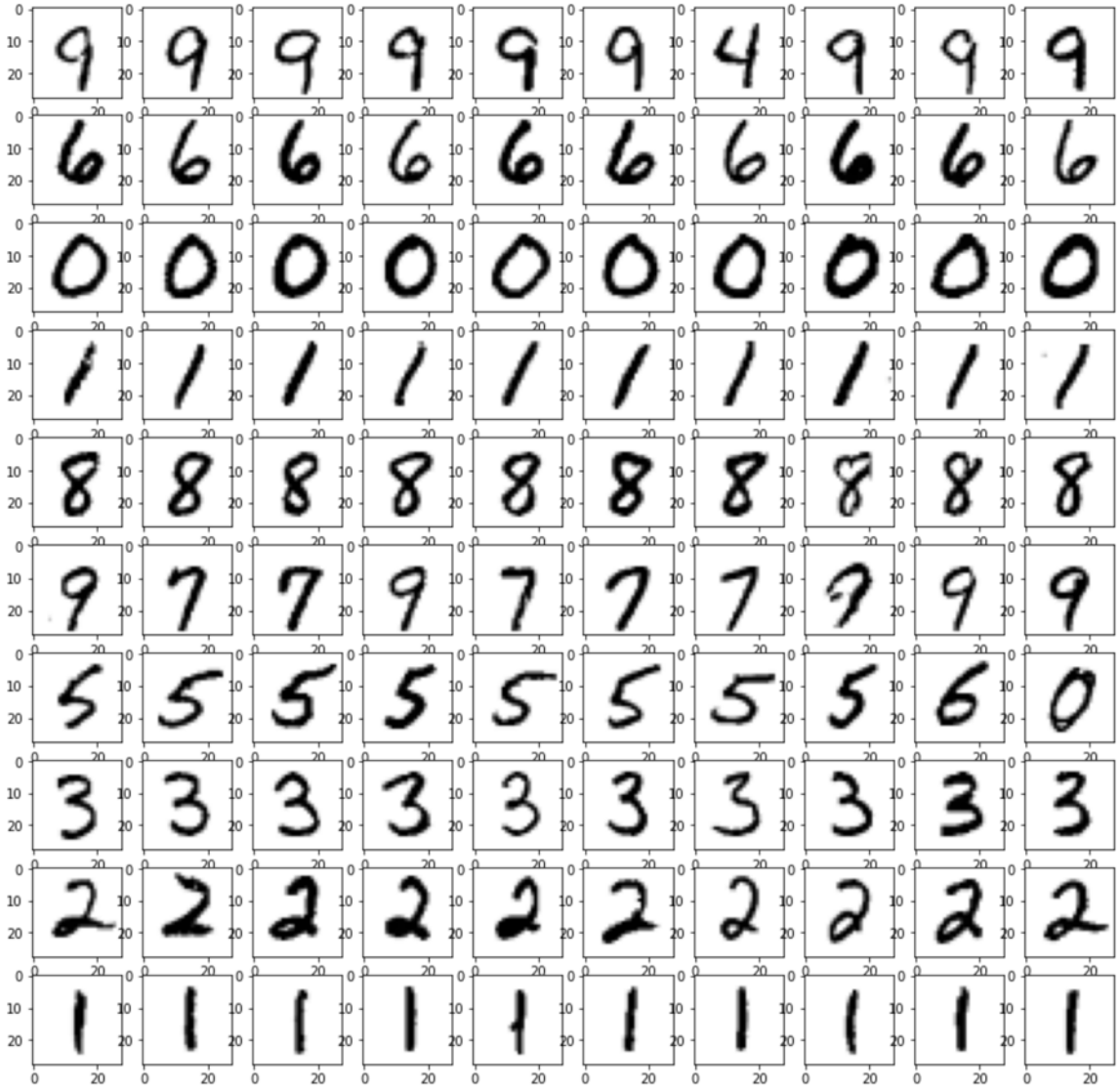
Plotting the images, eyeballing digits (P=20, K=10)

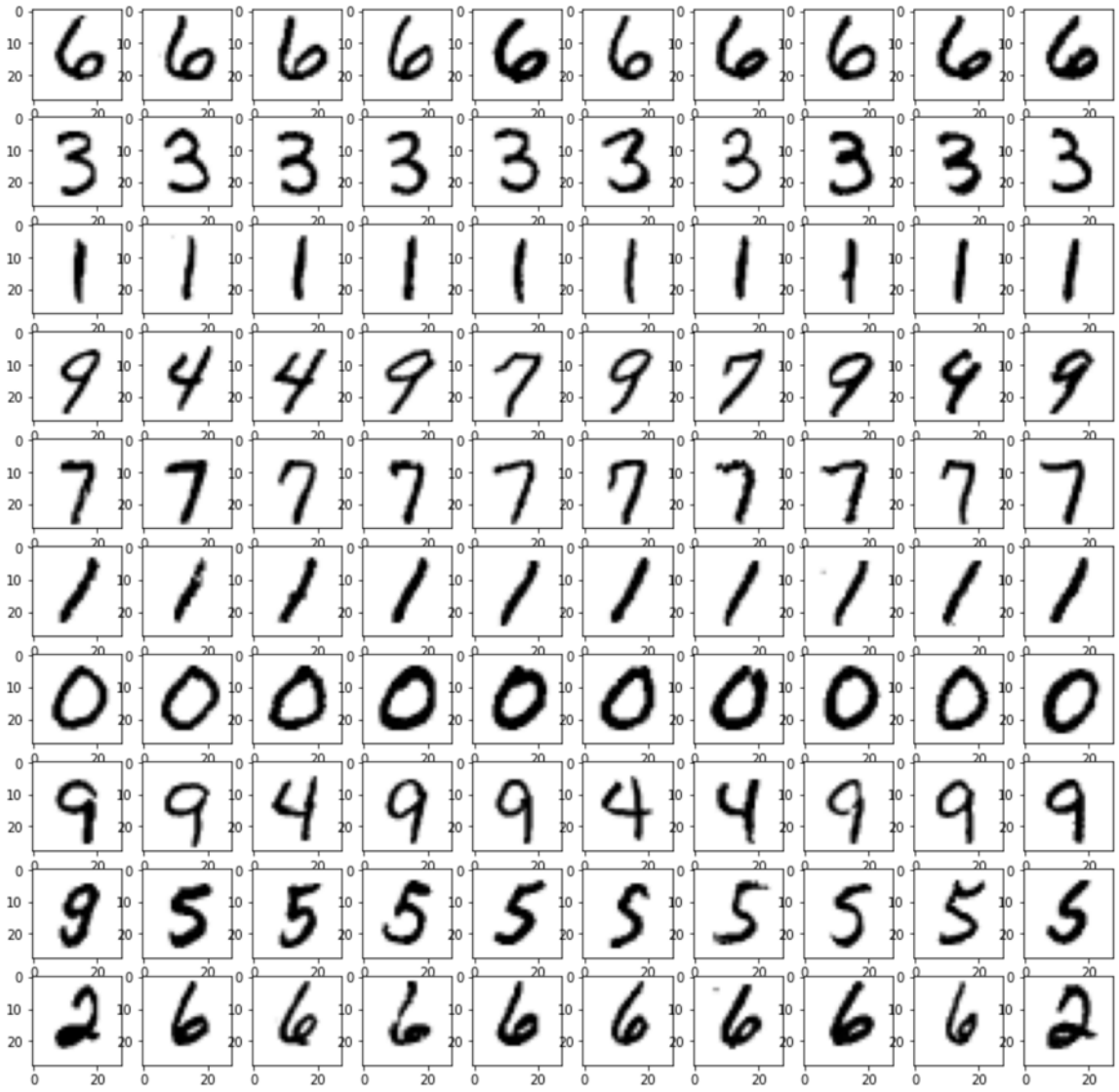
Using argsort on the negation of npmin10 and npmax10, I return the indices of the 10 smallest distances for each cluster. (Since argsort returns the max value I had to negate the matrix). I then plot the corresponding datapoints in a 10X10 figure for both max and min jclust runs. Each row indicates the Kth cluster and should have similar features via Kmeans algorithm.

In [205]:

```
temp=[]
fig, axs = plt.subplots(10, 10, figsize=(15,15))
for i in range(10):
    a = (-npmin10[:,i]).argsort()[-10:][::-1]
    temp.append(a) #gives index of 10 closest pictures each cluster
    print(temp[i])
for i in range(10):
    for j in range(10):
        axs[i,j].imshow(data_test[temp[i][j],:].reshape(28,28),cmap='binary')
    #print(data_test_label)

temp=[]
fig, axs = plt.subplots(10, 10, figsize=(15,15))
for i in range(10):
    a = (-npmax10[:,i]).argsort()[-10:][::-1]
    temp.append(a) #gives index of 10 closest pictures each cluster
    print(temp[i])
for i in range(10):
    for j in range(10):
        axs[i,j].imshow(data_test[temp[i][j],:].reshape(28,28),cmap='binary')
    #print(data_test_label)
```

```
[8623 1309  105 1165 3485  496 7669  773 3592 2983]
[6933 2458 5276 1373 3362 6261 9312 8779 8831 2759]
[4542 5659 4631 1723 8127 1333  592 7289 6201 8671]
[3211 9845 5461  202 6276 9802 6456 7091 6678 8159]
[9420 6275 3588 8585 3374 7313 5489 6612 1719 8618]
[5429 6267 5533 5414 6586 5363 9864 9638 9362 1745]
[2986 8940 6483 7351 3814 5480 8987 7352 3218  993]
[4770 2083 1605 4537  310 2634 1531 6863 6365 2303]
[ 208 3142 6193 7281 4296 6534 3661 4189 6217 3223]
[4931 3919 4774 5524 1994 1350 2541  675  663  342]
[5230 5884  138 6350  463 6824 9373 6354 6218 7078]
[4770 1605 3715 2076 2083 4537  310 5606  855 6863]
[4931 9291 3648 1350  675 2164 1238 1994 1867 1876]
[5247 5101 8917 6656 8202 9785 7238 7243 9517 9659]
[6586 2567 1401 6856 2275 3098 1321 1925 9448 9340]
[4262 3211 6468 7165 7567 5025 9030 8159 9485 6502]
[4542 8127 6201 8671 7289 6293 8730 4631 5659 5305]
[3485  105 7669 1309  496 3282 9406 8623 2323 2983]
[ 448 7324 2586 7003 7351 2525 3952 9331 9428 1637]
[4296 9334 9961 5743 6966 2026 8814 8399 1951 2662]
```





Generating a table:

Through eyeballing the data and comparing to the 2 rows of centroids from above. I have generated a 2X10 table for e ach jclust max and min runs below. They indicate the correctly and incorectly classified closest 10 data points. For each table, the top row signifies the correct amount of classifications and the bottom signifies the amount of wrongly classified. We can see that the worst performing group was group index5 for min with 5 misclassifications and the worst group for max was group index3 with 4 misclassified.

In [210]:

```
min_correct_table = np.zeros(10)
max_correct_table = np.zeros(10)
min_correct_table[0] = 9
min_correct_table[1] = 10
min_correct_table[2] = 10
min_correct_table[3] = 10
min_correct_table[4] = 10
min_correct_table[5] = 5
min_correct_table[6] = 8
min_correct_table[7] = 10
min_correct_table[8] = 10
min_correct_table[9] = 10
max_correct_table[0] = 10
max_correct_table[1] = 10
max_correct_table[2] = 10
max_correct_table[3] = 6
max_correct_table[4] = 10
max_correct_table[5] = 10
max_correct_table[6] = 10
max_correct_table[7] = 7
max_correct_table[8] = 9
max_correct_table[9] = 8

min_misclassified_table = 10 - min_correct_table
max_misclassified_table = 10 - max_correct_table
min_correct_table = min_correct_table.astype(int)
max_correct_table = max_correct_table.astype(int)
min_misclassified_table=min_misclassified_table.astype(int)
max_misclassified_table=max_misclassified_table.astype(int)
print("min:")
print(min_correct_table)
print(min_misclassified_table)
print("max:")
print(max_correct_table)
print(max_misclassified_table)
```

```
min:
[ 9 10 10 10 10  5  8 10 10 10]
[1 0 0 0 0 5 2 0 0 0]
max:
[10 10 10  6 10 10 10  7  9  8]
[0 0 0 4 0 0 0 3 1 2]
```

The tables above show for the jclust runs the amount of errors in classifications. The bottom row is the error and the top row is the correct classification.

Repeating for K =5 and P =10

In [212]:

```
#Running Main 10 times (P=10, K=15)
finaljclust = []
jclustaxes = []
iterationaxes = []
finalcentroids = []
for P in range(10):
    print("P = " +str(P+1)+" :")
    finalcentroid, finalcluster, jclust, jclustaxis, iterationaxis= main(data_test, 5)
    finaljclust.append(jclust)
    jclustaxes.append(jclustaxis)
    iterationaxes.append(iterationaxis)
    finalcentroids.append(finalcentroid)
```

```
P = 1 :
iteration 0: jclust:52.38178049177675
iteration 1: jclust:47.266768498206275
iteration 2: jclust:45.34965250685265
iteration 3: jclust:44.515781111936484
iteration 4: jclust:44.07985944083432
iteration 5: jclust:43.88792542953923
iteration 6: jclust:43.762049291340084
iteration 7: jclust:43.64944326165741
iteration 8: jclust:43.53583169695742
iteration 9: jclust:43.43433120808416
iteration 10: jclust:43.34668044884644
P = 2 :
iteration 0: jclust:52.336224428417104
iteration 1: jclust:47.9089195013115
iteration 2: jclust:46.48734488360789
iteration 3: jclust:45.47861722471101
iteration 4: jclust:44.877433228184664
iteration 5: jclust:44.50467185537248
iteration 6: jclust:44.134421662132084
iteration 7: jclust:43.89134248446654
iteration 8: jclust:43.81189231796681
P = 3 :
iteration 0: jclust:52.46171900075518
iteration 1: jclust:46.76701938016506
iteration 2: jclust:44.57742515369082
iteration 3: jclust:44.08668574802703
iteration 4: jclust:43.79651955319271
iteration 5: jclust:43.52573723714003
iteration 6: jclust:43.37095097078096
iteration 7: jclust:43.295631074741145
P = 4 :
iteration 0: jclust:52.26110550694543
iteration 1: jclust:46.60400508044507
iteration 2: jclust:45.14662940020975
iteration 3: jclust:44.59503071885417
iteration 4: jclust:44.19663150902353
iteration 5: jclust:43.910739896304186
iteration 6: jclust:43.71641639627461
iteration 7: jclust:43.59073872140681
iteration 8: jclust:43.506146505855966
P = 5 :
iteration 0: jclust:52.3995414386974
iteration 1: jclust:47.4090386975668
iteration 2: jclust:45.199184144189765
iteration 3: jclust:44.290185122893504
iteration 4: jclust:43.91680351241117
iteration 5: jclust:43.71098686371491
iteration 6: jclust:43.57723630206633
iteration 7: jclust:43.435484961161336
iteration 8: jclust:43.292397672570424
iteration 9: jclust:43.23109146785927
P = 6 :
iteration 0: jclust:52.34741927615721
iteration 1: jclust:47.467185040811046
iteration 2: jclust:45.478002467787555
iteration 3: jclust:44.857687447523894
iteration 4: jclust:44.318897098095924
iteration 5: jclust:44.13982851542181
iteration 6: jclust:44.09409248061958
P = 7 :
```

```
iteration 0: jclust:52.40145208144351
iteration 1: jclust:47.41057122612007
iteration 2: jclust:45.192013460303414
iteration 3: jclust:44.55048999316492
iteration 4: jclust:44.30409471965781
iteration 5: jclust:44.139371473598224
iteration 6: jclust:44.01973627998429
iteration 7: jclust:43.92770383087967
P = 8 :
iteration 0: jclust:52.37225331682096
iteration 1: jclust:46.89885033483845
iteration 2: jclust:45.199124327140844
iteration 3: jclust:44.812574717428575
iteration 4: jclust:44.58173619399317
iteration 5: jclust:44.40708184094221
iteration 6: jclust:44.241936683909486
iteration 7: jclust:44.08928772859367
iteration 8: jclust:43.929588910317435
iteration 9: jclust:43.75967794463625
iteration 10: jclust:43.627413746492
iteration 11: jclust:43.53055358814598
P = 9 :
iteration 0: jclust:52.438060670120144
iteration 1: jclust:47.57572127813068
iteration 2: jclust:45.27730537608077
iteration 3: jclust:44.568623587608656
iteration 4: jclust:43.96254482554906
iteration 5: jclust:43.577276680199525
iteration 6: jclust:43.387091158934744
iteration 7: jclust:43.28944511572398
P = 10 :
iteration 0: jclust:52.427664466194365
iteration 1: jclust:47.78156730160547
iteration 2: jclust:45.670919687446876
iteration 3: jclust:45.13069645702615
iteration 4: jclust:44.90943173355251
iteration 5: jclust:44.69863327700675
iteration 6: jclust:44.34212557494418
iteration 7: jclust:43.927601989949046
iteration 8: jclust:43.73419061271515
iteration 9: jclust:43.66893641822298
```

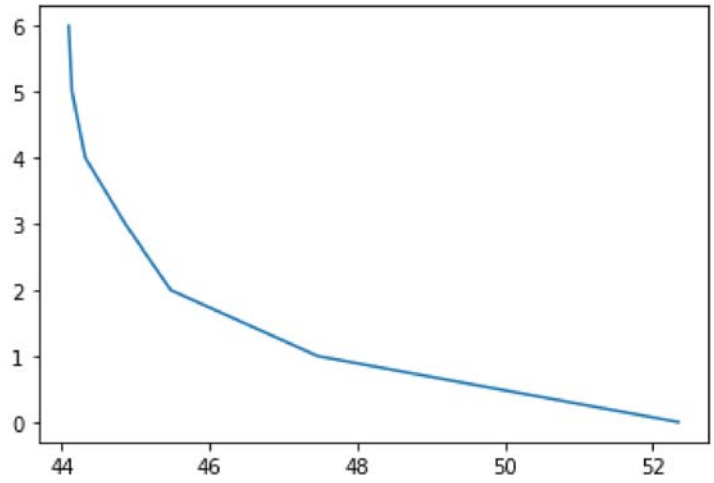
Results of P=10, K=5:

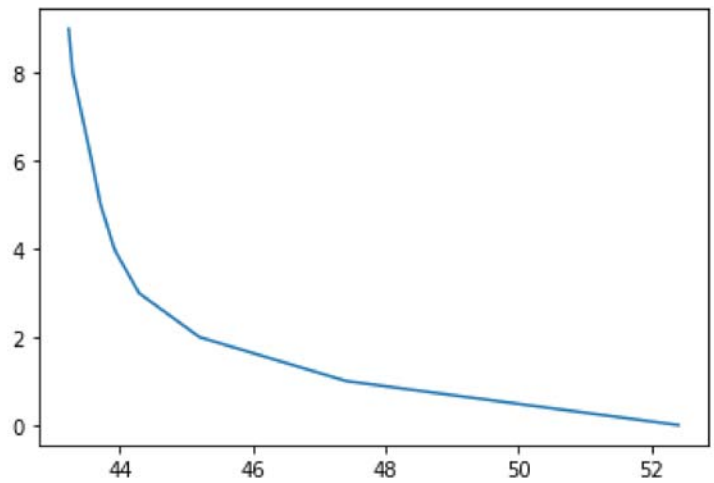
As computed from above, below, for the maximum and minimum JClust runs which are P=9 and P=10 respectively, corresponding to the 9th and 10th run respectively, has jclust values 44.09 and 43.43 respectively. The plots are as shown respectively. The top plot corresponds to the max run and the bottom plot corresponds to the min run.

In [213]:

```
# Retrieving Maxima and Minima
npfinaljclust=np.array(finaljclust)

maxjclust = max(finaljclust)
maxindex=np.argmax(npfinaljclust)
plotjclust(iterationaxes[maxindex],jclustaxes[maxindex])##
maxcentroid = finalcentroids[maxindex]

minjclust =min(finaljclust)
minindex=np.argmin(npfinaljclust)
plotjclust(iterationaxes[minindex],jclustaxes[minindex])##
mincentroid = finalcentroids[minindex]

print(maxindex)
print(minindex)
```

```
5
4
```

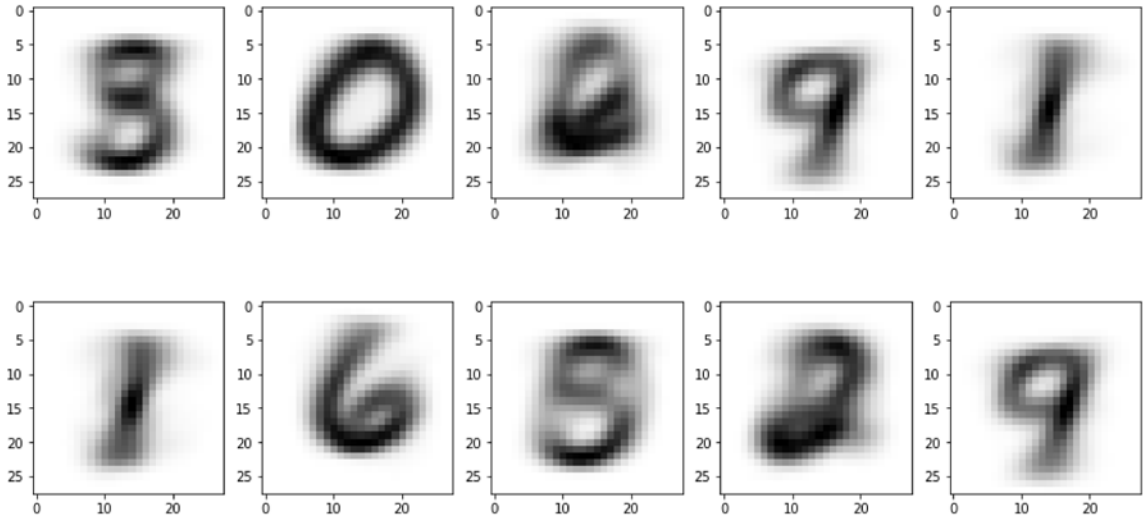




Below are the final group representatives for the max and min jclust runs respectively. The top row is max and bottom row is min.

In [214]:

```
K=5
maxcentroid = finalcentroids[maxindex]
mincentroid = finalcentroids[minindex]
fig = plt.figure()
fig.set_figheight(15)
fig.set_figwidth(15)
for i in range(K):
    fig.add_subplot(1,5, 1+i)
    plt.imshow(maxcentroid[i].reshape(28,28),cmap='binary')
    fig.add_subplot(2,5, 1+i)
    plt.imshow(mincentroid[i].reshape(28,28),cmap='binary')
```



Discussion of Results:

The majority of the centroids are quite distinguishable. With only 1 or 2 groups being blurry. This can have an affect on the classification accuracy later.

Finding closest 10 data points:

Below is the code that creates the numpy array npmin10, which stores the distances of every data point to every centroid.

In [215]:

```
K = 5
min10 = []
finalmin10=[]
for i in range(len(data_test)):
    distances = []
    for j in range(K):
        distances.append(euclideandist(data_test[i],maxcentroid[j]))
    min10.append(distances)#10,000 nparrays of20
npmax10 = np.array(min10)

min10 = []
finalmin10=[]
for o in range(len(data_test)):
    distances = []
    for f in range(K):
        distances.append(euclideandist(data_test[o],mincentroid[f]))
    min10.append(distances)#10,000 nparrays of20
npmin10 = np.array(min10)
```

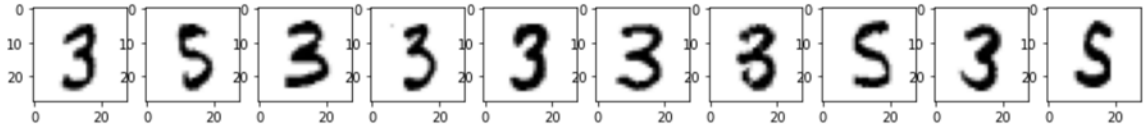
Plotting the images, eyeballing digits (P=10, K=5)

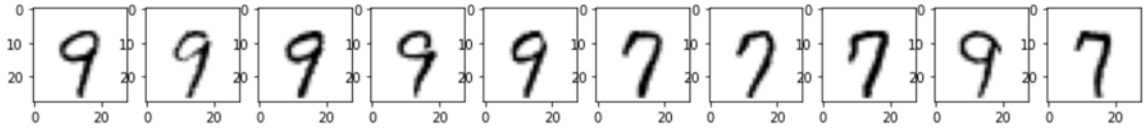
Using argsort on the negation of npmin10 and npmax10, I return the indices of the 10 smallest distances for each cluster. (Since argsort returns the max value I had to negate the matrix). I then plot the corresponding datapoints in a 10X10 figure for both max and min jclust runs. Each row indicates the Kth cluster and should have similar features via Kmeans algorithm.

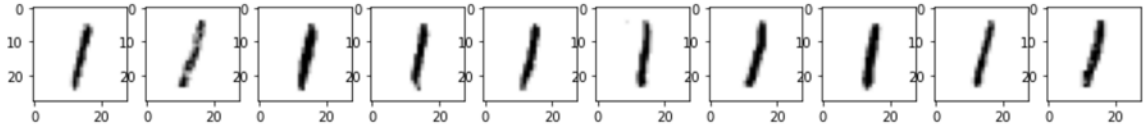
In [218]:

```
temp=[]
fig, axs = plt.subplots(5, 10, figsize=(15,15))
for i in range(5):
    a = (-npmin10[:,i]).argsort()[-10:][::-1]
    temp.append(a) #gives index of 10 closest pictures each cluster
    print(temp[i])
for i in range(5):
    for j in range(10):
        axs[i,j].imshow(data_test[temp[i][j],:].reshape(28,28),cmap='binary')
    #print(data_test_label)

temp=[]
fig, axs = plt.subplots(5, 10, figsize=(15,15))
for i in range(5):
    a = (-npmax10[:,i]).argsort()[-10:][::-1]
    temp.append(a) #gives index of 10 closest pictures each cluster
    print(temp[i])
for i in range(5):
    for j in range(10):
        axs[i,j].imshow(data_test[temp[i][j],:].reshape(28,28),cmap='binary')
    #print(data_test_label)
```

```
[2931 4971 6365 9882 7272 4818 2853  240 2549 2282]
[4542 5659 7289 6293 8127 4631 9519 6201 1723 8730]
[6966 9961 9334 2210 2458 1951 1578 1373 9269 9364]
[4901  471 5414 2544 9362 1401 3084 3098 2345 9448]
[3148 6618 6407 4859 1030 9291 2357 1876  835 6533]
[6618 3148 6407 9291 4859 1876 1030  835 3019 6623]
[2458 1373 6933 9312 6261 8831 3362 2759 5276 8779]
[4971  240 3917  395 3957 6624  153 5346 1967  890]
[4296 4189 4684 2981 2209 2662 2848 5823 6217 6607]
[4901  471 2544 5414 3247 9362 2345 8623 1401 3098]
```







Discussion of results:

For both the index and the image plots, the first 5 rows are for the minimum run and the last 5 rows are for the maximum run. Each row has 10 images corresponding to the 10 nearest datapoints. Comparing the the group representatives, I will generate a table below.

Generating max and min tables:

In [219]:

```
min_correct_table = np.zeros(5)
max_correct_table = np.zeros(5)
min_correct_table[0] = 7
min_correct_table[1] = 10
min_correct_table[2] = 10
min_correct_table[3] = 6
min_correct_table[4] = 10
max_correct_table[0] = 10
max_correct_table[1] = 10
max_correct_table[2] = 7
max_correct_table[3] = 9
max_correct_table[4] = 7

min_misclassified_table = 10 - min_correct_table
max_misclassified_table = 10 - max_correct_table
min_correct_table = min_correct_table.astype(int)
max_correct_table = max_correct_table.astype(int)
min_misclassified_table=min_misclassified_table.astype(int)
max_misclassified_table=max_misclassified_table.astype(int)
print("min:")
print(min_correct_table)
print(min_misclassified_table)
print("max:")
print(max_correct_table)
print(max_misclassified_table)
```

```
min:
[ 7 10 10  6 10]
[3 0 0 4 0]
max:
[10 10  7  9  7]
[0 0 3 1 3]
```

Eyeballing and verifying through the label produces the results as seen above. The 2 tables correspond to the max and min jclust runs and their correct and incorrect classifications for the 10 nearest data points.

Comparison of the performance of the supervised binary classifier from Least Squares and the unsupervised classifier from Kmeans:

While the least squares classifier requires the labels to be able to train the data, the Kmeans method only requires the data. By not knowing even how many different classes there are, kmeans will still group similar data together and form a prediction scheme. We do know that the correct number of classes is K = 10 as there are 10 hand written digits.

Comparing the results of both classifiers, I believe the least squares classifier is more accurate as we can see from the low error rates computed. Kmeans classifier on the other hand can wrongly classified labels even for the 10 nearest datapoints. However, kmeans is powerful in the fact that it doesn't need a supervised amount of classes and is essentially "hands off". The more you iterate the data through the algorithm, the more solid the centroids become and hence more accurate. However, for a limited amount of data points or time, least sqaures is still more effective.