Programming Assignment 2: Clustering

Background/Intuition:

As we progress into the age of digital information, the amount of data available for us to work with is staggering. You can peruse a billion images through google search, or query millions of songs with Spotify's API. However, while this data is in abundance, **labels** for it are usually a lot harder to come by. Suppose you wanted to group songs into subjective categories of 'chill and relaxing' versus 'upbeat and energetic'.

You might process these songs and generate some potentially relevant quantifiers, say the average pitch of the song (lower = more bass, higher = more treble) and its average amplitude (lower = quieter, higher = louder).

Song	Average Pitch	Average Amplitude
Typhoons, Royal Blood	-18.1	19.6
Waltz No. 2, Shostakovich	-7.6	11.2
Sandstorm, Darude	16.2	18.6
...	...	...

Note: Not real data

But unless you're willing to pay some music-appreciators a lot of money to listen to and review every song (of potentially millions of songs), there's no easy way to get definitive labels of 'relaxing' versus 'upbeat' for this data. Instead we turn towards **unsupervised** learning.

If we can cluster these songs into groups of similar pitch and amplitudes, even without labels we might be able to learn a lot about their distribution. We could listen to five or six songs from a cluster of thousands to get a good understanding of what kind of music it represents. And if we have a labeled song, maybe your favorite upbeat track for when you're at the gym, we can look at what cluster it gets put into and explore if the other songs in the cluster are also good upbeat tracks for the gym (this is not unlike the way real music services give you recommendations based off your listening habits).

Now that we've hopefully sold you on the excitement of unsupervised clustering, let's write our own k-means implementation and run it on some real data.

Format of this PA:

1. Debugging Practice
2. K-Means Helper Functions
3. Run K-Means on Simulated Data

4. Run K-Means on MNIST Image Data

ACT 1: Before We Make Code, We Un-Break Code

1A: Sums and Prods

In the code below, we are trying to sum the columns of `X`, and then matrix multiply them with `y` to produce `Q` similar to the example below:

$$X = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

$$Q = \begin{bmatrix} x_{11} + x_{12} \\ x_{21} + x_{22} \end{bmatrix} \begin{bmatrix} y_1 & y_2 \end{bmatrix} = \begin{bmatrix} (x_{11} + x_{12})y_1 & (x_{11} + x_{12})y_2 \\ (x_{21} + x_{22})y_1 & (x_{21} + x_{22})y_2 \end{bmatrix}$$

```
import numpy as np

X = np.array([[1,2,3],
              [4,5,6],
              [7,8,9]])
y = np.array([[0, 2, 1]])

def broken_sum_prod(X, y):
    X_sum = np.sum(X, axis=1)
    Q = X_sum @ y
    return Q
```

But when we run this code, all we get is a `ValueError`, seems something went wrong with our dimensions.

```
broken_sum_prod(X,y)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-2-806ccf8bf4fd> in <module>
----> 1 broken_sum_prod(X,y)

<ipython-input-1-6aaf69815099> in broken_sum_prod(X, y)
      8 def broken_sum_prod(X, y):
      9     X_sum = np.sum(X, axis=1)
---> 10     Q = X_sum @ y
     11     return Q

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with
```

SEARCH STACK OVERFLOW

Investigate the shape of `X_sum` and see if you can find an argument in the documentation of np.sum that can fix this error.

```
# ACT 1A: fix this code
def fixed_sum_prod(X, y):
    X_sum = np.sum(X, axis=1, keepdims = True)
    print(np.shape(X_sum))
    print(np.shape(y))
    Q = X_sum @ y
    return Q


fixed_sum_prod(X,y)
```

```
(3, 1)
(1, 3)
array([[ 0, 12,  6],
       [ 0, 30, 15],
       [ 0, 48, 24]])
```

1B: Take A Long Axis

It's often best to correct errors before we make them. Later in this assignment we will be interested in using the function np.take_along_axis to extract elements at desired indices from a matrix. You might want to familiarize yourself with its documentation before proceeding.

Here we attempt to use `indices` to index into `arr` to retrieve the values `[[1],[6],[8]]`; the 0th, 2nd, and 1st values in the respective rows.

```
arr = np.array([[1,2,3],
                [4,5,6],
                [7,8,9]])
indices = np.array([0, 2, 1])


np.take_along_axis(arr, indices, axis=1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-e0d8845f74ee> in <module>
----> 1 np.take_along_axis(arr, indices, axis=1)

<__array_function__ internals> in take_along_axis(*args, **kwargs)
```

1 frames

We appear to have ran into a dimension error, lets look at the shapes of `arr` and `indices` and investigate.

```
print(arr.shape)
print(indices.shape)
```

```
(3, 3)
(3,)
```

Looking back at the documentation we spot that for the `indices` variable there's a descriptor `"This must match the dimension of arr, but dimensions Ni and Nj only need to broadcast against arr."`. Though the latter part sounds cryptic, the beginning (`must match the dimension of arr`) is pretty clear. Our `arr` is two dimensional and our `indices` are a 1D vector, let's try fixing that.

```
indices = np.array([[0, 2, 1]])
print(indices.shape)
```

```
(1, 3)
```

```
np.take_along_axis(arr, indices, axis=1)
```

```
array([[1, 3, 2],
       [4, 6, 5],
       [7, 9, 8]])
```

Hmm, this is strange; we seem to be taking the 0th **and** the 2nd **and** the 1st element of each row, something still isn't right. See if you can figure out what's going on, and get the function call to return just `[[1],[6],[8]]`.

```
# ACT 1B: fix this code
indices = np.array([[0], [2], [1]])
np.take_along_axis(arr, indices, axis = 1)
```

```
array([[1],
       [6],
       [8]])
```

What was the error? Do we now have a better understanding of `dimensions Ni and Nj only need to broadcast against arr`? (think about how the dimensions of `indices` affect the output)

▼ Your answer here:

The error was with how we constructed the *indices* array. Since our dimensions were originally (1,3) we broadcasted to the three rows since we only had one row vector. When we change the *indices* array to a (3,1) array, we no longer need to broadcast to match the rows of arr and we can get our desired result.

Clustering Overview

In this assignment we'll implement k-means clustering. Let's call our input data X, the cluster centers W, and the number of clusters k. Each row of X is a data point, each row of W is the coordinates of a cluster center (a centroid). The goal of this algorithm is to minimize the distance between every data point and its closest centroid.

In this k-means implementation, our loss is **the mean of square euclidean distances** between each point and its closest centroid. More precisely, if our centroids are $\mathbf{w}_1, \ldots, \mathbf{w}_k$ and our points are $\mathbf{x}_1, \ldots, \mathbf{x}_n$, then our loss (averaged over n points) is:

$$\frac{1}{n}\sum_{i=1}^{n}\min_j ||\mathbf{x}_i - \mathbf{w}_j||^2,$$

where the $\min_j$ just means we incur loss only to the closest centroid j, and discard the distance to any centroids that aren't j.

As we saw in lecture this objective can also be written as:

$$\frac{1}{n}\sum_{j=1}^{k}\sum_{i\in C_j}\left\|\mathbf{x}_i - \mathbf{w}_j\right\|^2$$

Where C_j is the set of points belonging to cluster j, and k is again the total number of clusters.

Note: This loss is **not exactly the same** as the one covered in lecture, which minimized the sum of square distances rather than the mean. We opt to use the mean of the distances so that we can later compare training and validation losses for different numbers of datapoints.

To implement this in code, you will create a method to compute pairwise distances between data points and centroids, along with a method that re-estimates the centroids after each step. We'll

need a way to associate which point belongs to which centroid (which we'll achieve through a method `associate()`), and lastly a method to randomly initialize the centroids.

Implementing K-means

ACT 2: Step-by-step for `compute_sq_dists`

For `compute_sq_dists`, your goal is to find the squared distance between each point and each centroid. We ultimately want a 2-D array where the (i, j)-th entry is the squared distance between $\mathbf{x}_i$ and $\mathbf{w}_j$. **Looping through each example in X is too slow, so we'll do this with vectorized Numpy operations**.

Let $\mathbf{x}_i$ and $\mathbf{w}_j$ be the i-th and j-th rows of X and W respectively. We'd like to find the squared distance between each point and each cluster centroid: this helps compute the square loss, and to find the closest centroid we will just choose the centroid with the smallest square distance. Now, a cool math trick! It turns out that:

$$||\mathbf{x}_i - \mathbf{w}_j||^2 = ||\mathbf{x}_i||^2 + ||\mathbf{w}_j||^2 - 2\mathbf{x}_i \cdot \mathbf{w}_j$$

To find the squared distance between each point and each centroid, first we compute the squared norm of each row of X (corresponding to $||\mathbf{x}_i||^2$ in the above equation) and the squared norm of each row in W ($||\mathbf{w}_j||^2$). We can make a matrix where the (i, j)-th entry is $||\mathbf{x}_i||^2 + ||\mathbf{w}_j||^2$ by summing a column of X's row-norms with a row of W's row-norms. Recall that python + Numpy will automatically perform broadcasting if one is a column vector and the other a row vector.

To find $\mathbf{x} \cdot \mathbf{w}$, we multiply XW^T, which results in an array where the in (i, j)-th entry equals $\mathbf{x}_i \cdot \mathbf{w}_j$, the inner product of the i-th row of X with the j-th row of W.

Reminder: no for loops.

```
#ACT 2 squared distance_matrix.  Use the trick described above to compute the distance
def compute_sq_dists(X, W):
    '''
    Inputs:
    X is a 2-D Numpy array of shape (n, d), each row is a data point
    W is a 2-D Numpy array of shape (k, d), each row is a cluster centroid

    Output:
    2-D Numpy array of shape (n, k) where the i,j-th entry represents the squared eucl
    from the ith row of X to the jth row of W.
    '''

    X_sq = np.sum(np.square(X), axis = 1, keepdims = True)
```

```
    X_sq = np.sum(np.square(X), axis = 1, keepdims = True)
    W_sq = np.sum(np.square(W), axis=1, keepdims = True)

    sq_dists =  X_sq+W_sq.T
    sq_dists -= 2*X@W.T

    # your code here
    return sq_dists
```

ACT 3: Tips for `update_centroids`

Given data points `X` and vector `assoc`, an array of length `n` with containing values ranging from `0` to `k` which indicates which data point is *associated* to what cluster, our goal is to return a matrix of `k` new centroids, where the value of each centroid is the mean of its associated points.

Using the notation from lecture, the new centroid location for points belonging to cluster C_j is:

$$\mathbf{w}_j = \overline{\mathrm{x}}\left(\mathrm{C_j}\right) = \frac{1}{\mathrm{m_j}} \sum_{\mathrm{i} \in \mathrm{C_j}} \mathbf{x}_i \qquad \left(\mathrm{m_j} = \left|\mathrm{C_j}\right|\right)$$

In this act (**and only in this act**) it is okay to loop over `k`, the number of clusters, as this can reduce the complexity of the code. You're still encouraged to solve this without loops, and cannot loop over the number of data points `n` as this will grow prohibitively slow for large `n`.

```
#ACT 3 Implement update_centroids, computing the means of our most recent clusters
def update_centroids(X, k, assoc):
    '''
    Inputs:
    X is a 2-D Numpy array of shape (n, d), each row is a data point
    k is the number of clusters
    assoc is a 1-D array of size n indicating the center closest to each point

    Output:
    A 2-D array of cluster centroids size (k,d), where each row is a centroid.
    If there are no points associated with a cluster, this should return
    all zeros for that row/cluster.
    '''

    W = [[0]*X[0].size]*k
    for k_val in range(k):
      assoc_dupe = assoc
      #print("new for iter, value k = ", k_val)
      # get array of indices that match value k
      # https://stackoverflow.com/questions/6294179/how-to-find-all-occurrences-of-an-
      #np.where(assoc_dupe == k_val, k_val, 0)
      #print(assoc_dupe)
      k_arr = [i for i,val in enumerate(assoc_dupe) if val==k_val]
      # for each index that matches k, get those arrays
```

```
    # https://stackoverflow.com/questions/22927181/selecting-specific-rows-and-colur
    centroid_arr = X[k_arr, :]
    # https://numpy.org/doc/stable/reference/generated/numpy.matrix.sum.html
    centroid_arr = centroid_arr.sum(axis=0)

    # np.stack(W, centroid_arr)
    W[k_val] += centroid_arr / len(k_arr)
    #print(W[k_val])


  # your code here
  W = np.vstack(W)
  return W

  a = np.arange(20).reshape((5,4))
  #print(a)
  k=2
  assoc = [0,1,1,0,0]

  for x in range(7): print(x, end=" ")

  print(update_centroids(a,k,assoc))

  #x = np.matrix([[1, 2], [4, 3]])
  #x.sum()
  #x.sum(axis=0)

 0 1 2 3 4 5 6 new for iter, value k =  0
 [ 9.33333333 10.33333333 11.33333333 12.33333333]
 new for iter, value k =  1
 [6. 7. 8. 9.]
 [[ 9.33333333 10.33333333 11.33333333 12.33333333]
  [ 6.          7.          8.          9.        ]]
```

Act 4: Find centers associated with each data point given distances from each point to each center.

In `associate_centroids` we will find which centroid each data point is closest to, and also return the loss for convenience. W should first compute the square distances to each point, using a function we defined above, then use that to create the `assoc` vector as defined above/below. This can then be used to compactly estimate our overall loss (copied here for convenience, **reminder**: this is not exactly the lecture loss, we're taking the average over the number of points n here).

$$\frac{1}{n}\sum_{j=1}^{k}\sum_{i\in C_j}\left\|x^i-\bar{x}\left(C_j\right)\right\|^2$$

For computing the loss, you might need np.take_along_axis from earlier.

```
#ACT 4 Using the distance matrix find centers associated with each point and compute l
def associate_centroids(X, W):
    '''
    Inputs:
    X is a 2-D Numpy array of shape (n, d), each row is a data point
    W is a 2-D Numpy array of shape (k, d), each row is a cluster centroid

    Output:
    (loss, assoc) where
    assoc is a 1-D array of size n indicating the center closest to each point
    loss is the average loss of the data from those associations.
    '''

    #### ACT 4A Compute the *indices* `assoc` corresponding to minimum entry in each r
    sq_dists = compute_sq_dists(X,W)
    assoc = np.expand_dims(np.argmin(sq_dists, axis=1), axis=1)

    ###   ACT4B ### Use `assoc` to get values from X and calculate loss
    loss = np.sum(np.take_along_axis(sq_dists, assoc, axis=1))
    assoc1 = np.squeeze(assoc)
    return loss, assoc1


    a = np.arange(20).reshape((5,4))
    print("a", a)
    c = np.arange(3, 11).reshape((2,4))
    print("c", c)


    print("ans:", associate_centroids(a,c))
```

```
a [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]
c [[ 3  4  5  6]
 [ 7  8  9 10]]
ans: (468, array([0, 0, 1, 1, 1]))
```

Act 5: Implement random initialization of centroid

To start running k-means we need starting points to use as centroids. Use np.random.choice to select `k` random data points from `X` to use as initial centroids. We want to make sure our selected initialization points are *unique*, i.e. we don't accidentally pick the same point twice; np.random.choice should have an argument that guarantees this.

```
# ACT 5 random initialization
def rand_init(X,k):
    '''
    Inputs:
    X is a 2-D Numpy array of shape (n, d), each row is a data point
    k is the number of clusters
    Outputs:
    C a 2-D Numpy array of shape (k, d) of random points at which to initialise centro
    '''
    rows = np.random.choice(len(X), k, replace = False)
    C = X[rows, :]


    return C

    a = np.arange(20).reshape((5,4))
    print("a", a)

    print(rand_init(a,2))
```

```
 a [[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]
  [12 13 14 15]
  [16 17 18 19]]
 [[12 13 14 15]
  [16 17 18 19]]
```

Act 6: Putting everything together.

Implement the main iteration loop, alternating between associating data points to their nearest center, and estimating new centers. Remember to keep track of the loss at each iteration, and add an epsilon stopping criterion (if the loss hasn't decreased by more than epsilon since the previous iteration -- proportional to the current loss -- break the loop).

For more clarification, your stopping criterion should for iteration t check if:

$$\frac{\text{loss}_{t-1} - \text{loss}_t}{\text{loss}_t} < \epsilon$$

```
# ACT 6 implement the K-means loop
def k_means(X, k, max_iters=50, eps=1e-5):
    '''
    Perform k-means clusting on X using k clusters, running for max_iters loops,
    and stop early if loss decreases by a proportion less than eps.
    Early stopping does not occur before epoch 2.

    Output (W, loss) the final centers W and each iteration's loss
    '''
    #course notes, page 77
```

```
    prev_loss = 0
    W = rand_init(X, k)
    for i in range(max_iters):
      loss, C = associate_centroids(X,W)
      Y = update_centroids(X, k, C)
      if prev_loss - loss < eps:
        if i>=2:
          break
      prev_loss = loss

    return W, loss
```

K-means Scatter Plot Helper Function

This has been written for you.

- Read the code and understand it from a high-level point of view.
- Understand how to call this function

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.linalg import eigh
np.random.seed(324)

# Purpose:
# --------
# Plot d-dimensional sample in 2D with cluster indicators (uses PCA to lower dimensior
#
# How to call:
# ------------
# X is the examples: each row is an example
# C is a 1-d array of associated centroid index (integers):
#    Each entry in C corresponds to a row in X, so len(C) = X.shape[0]
#    OR
#    C = [] means only data visualization w/o centroids
#
#
# Additional paramters (do not change unless instructed):
# bb is the bounding box
# `colors` indicates the colors: the default colors the
#     first cluster `b` or blue, next cluster `g` or green, etc.
#
def plot_sample(X, C, bb=[], ms=10, colors='bgrcmk'):
    plt.figure(figsize=(12,6))
    plt.subplot(1,2,1)

    from itertools import cycle
    if X.shape[1] > 2:
        err_str = "X contains {}-dimensional points. Data points must be 2-dimensional
```

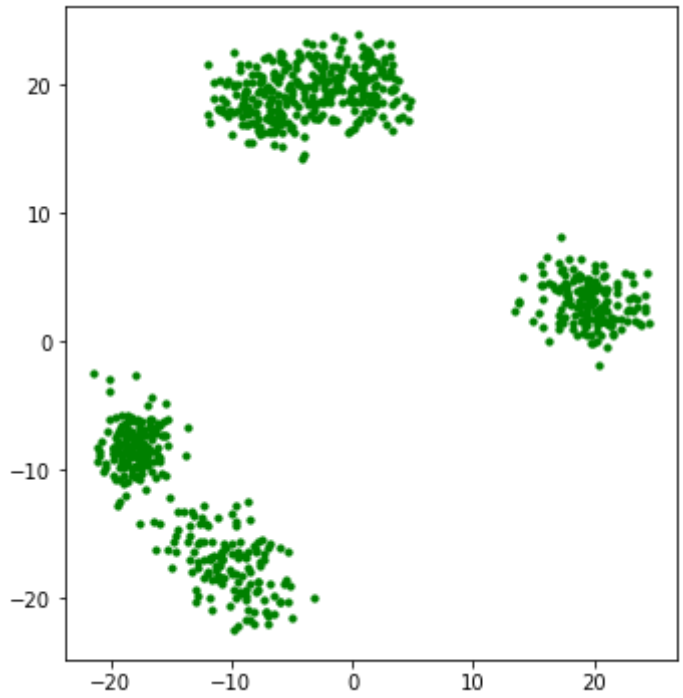
```
            raise Exception(err_str)
        if len(C) == 0: C = np.ones(X.shape[0])
        k = int(C.max()) + 1
        if bb != []:
            plt.xlim(bb[0]), plt.ylim(bb[1])
        cycol = cycle(colors)
        for i in range(k):
            ind = C == i
            col = next(cycol)
            plt.scatter(X[ind,0], X[ind,1], s=ms, c=col)
        plt.show()
```

Experimenting With Synthetic Data

Now that we've implemented k-means clustering, let's run an experiment on simulated data. First let's visualize the data (something you should get in the habit of):

```
Xt = np.load("simulated_data.npy")
N = len(Xt) # number of samples
split = int(0.8*N) # keep 80% for train, 20% for val

Xt_train = Xt[0:split,:]
Xt_val= Xt[split:,:]
_ = plot_sample(Xt_train, [])
```



How many clusters does it look like the data can/should be split into? Why did you pick that number?

(Optional, what could these clusters be if these points corresponded to our song data as described at the start?)

Your answer here:

The data looks like it should be split into four clusters. There are two distinct clusters in the top middle and middle right of the graph. In the bottom left of the graph, I see a close cluster around (-20, -10) and a looser cluster around (-10,-20). As such, it seems like there should be four clusters. However, the two clusters in the bottom left are close to each other, so there is a possibility that they should be put in the same cluster and we end up with three clusters.

ACT 7: running k-means on simulated data

Run your k-means function on the training data `Xt_train`, report the loss vector and use `plot_sample` to plot the clusters.
Run each cell multiple times to see how the random initialization changes the final clusters.
Name your vector of associated centroids `assoc`, this will be relevant later

```
# ACT7a: Run k-means clustering on Xt_train, using k = 2 cluster
assoc, loss = k_means(Xt_train, 2)
print("assoc:", assoc)
print("loss:", loss)
# your code here
```

```
assoc: [[-18.41496  -8.66967]
 [-11.00102  20.29911]]
loss: 255612.8992353038
```

```
# ACT7b: Run k-means clustering on Xt_train, using k = 3 clusters
assoc, loss = k_means(Xt_train, 3)
print("assoc:", assoc)
print("loss:", loss)
# your code here
```

```
assoc: [[-20.94194  -8.51802]
 [  0.21424  21.4799 ]
 [ 13.82656   3.22689]]
loss: 50013.6590068643
```

```
# ACT7c: Run k-means clustering on Xt_train, using k = 4 clusters
# you might need to run this one several times to get the `expected` clusters
assoc, loss = k_means(Xt_train, 4)
```

```
print("assoc:", assoc)
print("loss:", loss)
# your code here
```

```
assoc: [[-11.17963 -18.24542]
 [-15.47097  -4.76817]
 [  4.54972  18.30534]
 [-20.26989  -6.94693]]
loss: 110093.16508832348
```

ACT 8: Loss Curves

Let's look at how our final loss changes as a function of `k`, the cluster count (i.e. let's see those elbow curves). For each `k` value in the range of `1,...,20` run k-means `20` times (to average out the variability from the initialization). Record the mean training and validation loss for each set of runs.

Finally, make a **well-labeled** plot of training and validation losses versus `k`.

Hint: you should never be running `k_means` on the validation dataset, *another function* you defined above can return you the validation loss given the centroid locations you learned from the training set

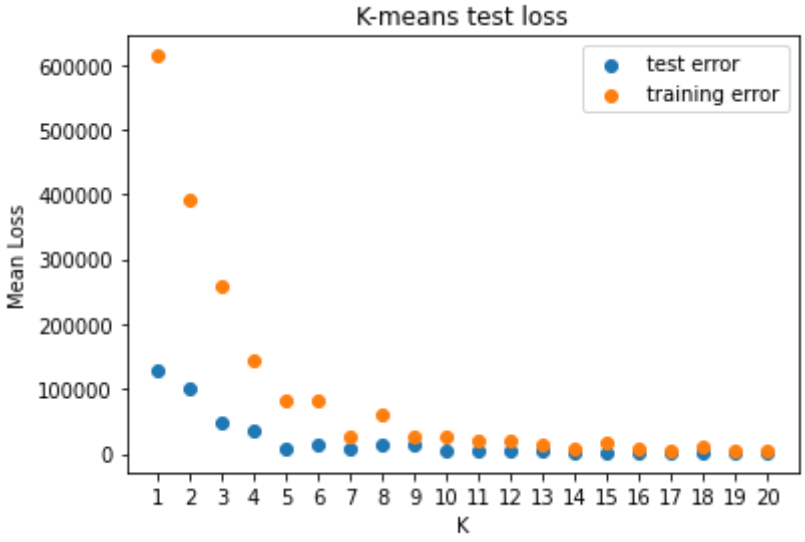
```
# Act8: build elbow curve of train + val losses
# Warning, this code should take 20-30 seconds to run; if it's taking forever, you pro
# vectorize your k-means code correctly (remember, no for loops over the data points)
losst = []
lossv = []
for k in range(1, 21):
  losst_k = 0
  lossv_k = 0
  for i in range(20):
    assoc, losst_ki = k_means(Xt_train, k)
    losst_k+=losst_ki
    lossv_ki, _ = associate_centroids(Xt_val, assoc)
    lossv_k+=lossv_ki

  losst.append(losst_k/20)
  lossv.append(lossv_k/20)
print("training loss:", losst)
print("test loss:", lossv)
```

```
# https://www.adamsmith.haus/python/answers/how-to-create-an-array-of-numbers-1-to-n-i
a_list = list(range(1,21))
plt.scatter(a_list, lossv, label = "test error")
plt.scatter(a_list, losst, label = "training error")
plt.xlabel('K')
```

```
plt.xticks(a_list)
# https://stackoverflow.com/questions/19125722/adding-a-legend-to-pyplot-in-matplotlib
plt.legend(loc="upper right")
plt.ylabel('Mean Loss')
plt.title('K-means test loss')
```

```
training loss: [613837.2922302216, 392986.8418166808, 257573.373881095, 142956.05
test loss: [128386.08282875639, 101076.605518421, 48265.81578205277, 35389.03117
Text(0.5, 1.0, 'K-means test loss')
```



Around how many clusters do we see the *elbow* in the loss curve (look back to the lecture slides if you need a reminder on what the elbow is).
Does this make sense given your previous observations on the distribution of the data?

▼ Your answer here:

The elbow seems to be around the six cluster mark. Since my prediction was four clusters, it's a bit more detail than my eyeballing concluded. However, it still makes sense, as some of the clusters in the four model could be better captured as two clusters (for example, the cluster at the top middle)

It looks like we can minimize both the validation and training loss by taking more and more clusters. Why would this be a bad idea in practice? Does having a ton of clusters likely **reduce** or **increase** our ability to infer useful information from the data?

Minimizing the validation and test loss simply by taking more clusters is a bad idea in practice because the objective of clustering is to get groups of similar objects. By having a ton of clusters,

we reduce the information we can infer across a cluster. For example, instead of saying that one cluster is country in a reasonable cluster model, a model with a ton of clusters might have a cluster that includes country songs made in year 2002 by a female artist whose name ends in a Y. There is more value in the lesser amount of clusters compared to a ton with regards to infering useful information from the data.

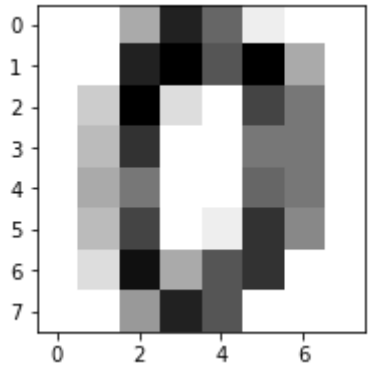
Experimenting With MNIST Data

Now that we understand some of the behavior of K-means for our simulated dataset, let's see how this extends to images of digits.

```
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA


#Load the digits dataset
digits = load_digits()

#Display the first digit
plt.figure(1, figsize=(3, 3))
plt.imshow(digits.images[0], cmap=plt.cm.gray_r, interpolation='nearest')
plt.show()
```



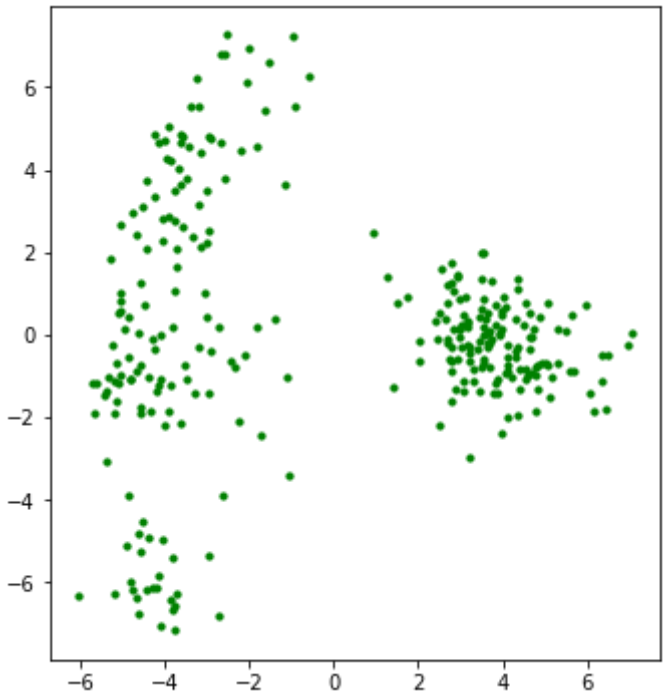

```
# load digits zero and one
X_digits, y_digits = load_digits(n_class=2, return_X_y=True)
Xt = scale(X_digits)
N = len(Xt)
split = int(0.8*N) # keep 80% for train, 20% for val

Xt_train = Xt[0:split,:]
Xt_val= Xt[split:,:]


# use PCA to reduce the data dimension to 2 for plotting
reduced_data = PCA(n_components=2).fit_transform(Xt_train)
```

Let's visualize this PCA-reduced data:

```
_ = plot_sample(reduced_data, [])
```



Does this data look *clusterable*? How many clusters do we expect? Why does this make sense given the data we loaded above?

Your answer here:

The data does look somewhat clusterable, but not super clean-cut. We would expect two clusters (one to the left of the graph, one to the right). This makes sense because we are uploading data that are either 0s or 1s in digits, so there are two 'real' clusters.

ACT 9: K-means on MNIST data

Run your k-means clustering code on this data and plot the clusters.

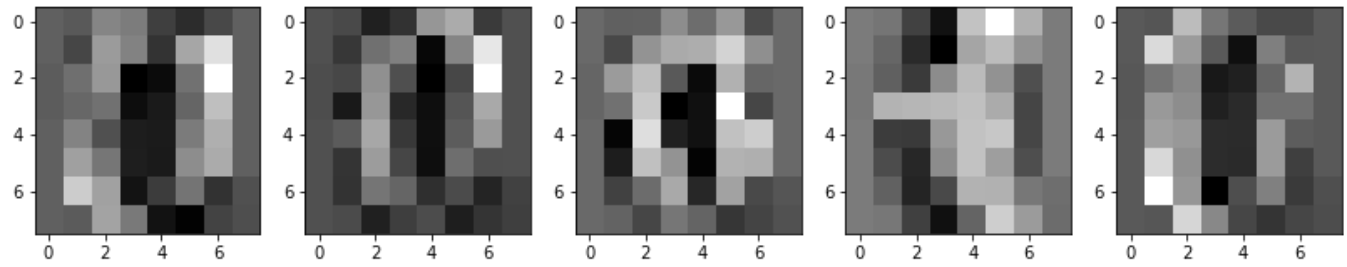
```
# ACT9: Run k-means on MNIST data with k=2
W, loss = k_means(Xt_train, 2)
_, assoc = associate_centroids(Xt_train,W)
```

Lets plot some representative samples from this clustering:

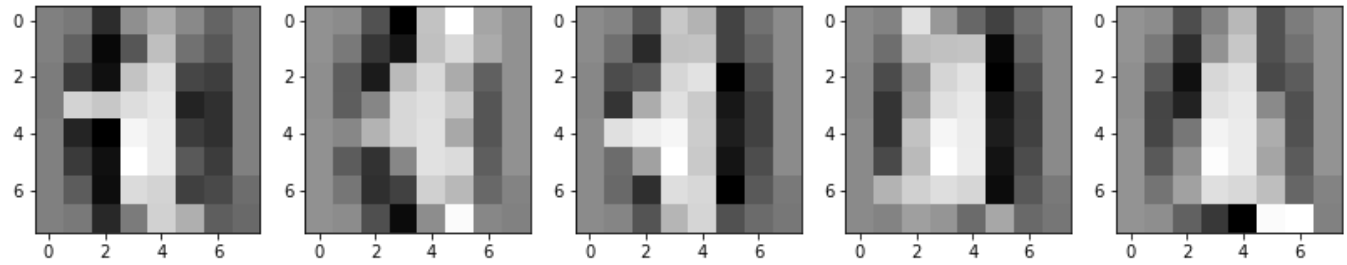
```
cluster_0 = Xt_train[np.where(assoc==0)]
cluster_1 = Xt_train[np.where(assoc==1)]

#print(Xt_train)
#print(cluster_0)


fig, axs = plt.subplots(1, 5)
fig.set_size_inches(14,14)
for i, ax in enumerate(axs):
    ax.imshow(cluster_0[i].reshape(8,8), cmap="gray")
```



```
fig, axs = plt.subplots(1, 5)
fig.set_size_inches(14,14)
for i, ax in enumerate(axs):
    ax.imshow(cluster_1[i].reshape(8,8), cmap="gray")
```



Do the images in each cluster look similar? If the code is working correctly they hopefully should! Note that the zero-th cluster might not correspond to the digit zero. If you were to use this system for digit classification, you'd manually have to look at some representatives of each cluster before, label them yourself, and then apply those labels to the rest of the data based on which cluster they belong to (this is still a lot more efficient than manually labeling a thousand numbers though).

ACT 10: More Elbow Curves

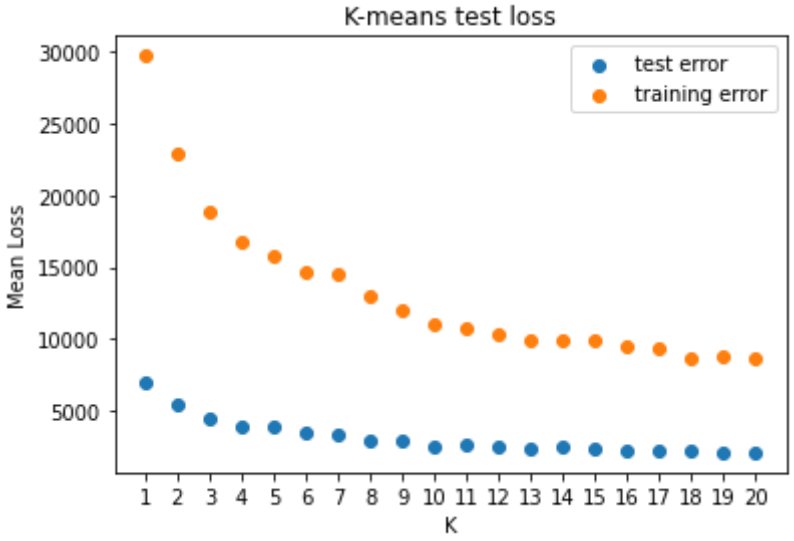
Copy your code from ACT 8, and plot train and validation loss curves for this MNIST data.

```
# Act10: build elbow curve of train + val losses
losst = []
lossv = []
for k in range(1, 21):
  losst_k = 0
  lossv_k = 0
  for i in range(20):
    assoc, losst_ki = k_means(Xt_train, k)
    losst_k+=losst_ki
    lossv_ki, _ = associate_centroids(Xt_val, assoc)
    lossv_k+=lossv_ki

  losst.append(losst_k/20)
  lossv.append(lossv_k/20)
print("training loss:", losst)
print("test loss:", lossv)


# https://www.adamsmith.haus/python/answers/how-to-create-an-array-of-numbers-1-to-n-i
a_list = list(range(1,21))
plt.scatter(a_list, lossv, label = "test error")
plt.scatter(a_list, losst, label = "training error")
plt.xlabel('K')
plt.xticks(a_list)
# https://stackoverflow.com/questions/19125722/adding-a-legend-to-pyplot-in-matplotlib
plt.legend(loc="upper right")
plt.ylabel('Mean Loss')
plt.title('K-means test loss')
```

```
training loss: [29679.992410369217, 22845.83489253793, 18907.011097617607, 16749
test loss: [6921.541109232423, 5461.204828372273, 4443.5917068106255, 3920.00621
Text(0.5, 1.0, 'K-means test loss')
```



Do the loss curves above go down as the number of clusters increase? Despite this, why would it

▾ Your answer here:

```
The loss curves above do go down as the nu
with digit classification we only need two
two digits. Additional ones will give us a
of digit (for example, some people put a l
However, those types of distinctions are f
sense to make a large value of K, as the a
importance.
```

The loss curves above do go down as the number of clusters increase. However, with digit classification we only need two clusters - one each to classify the two digits. Additional ones will give us a bit more clarity of different types of digit (for example, some people put a line in their zero, others do not). However, those types of distinctions are few in number. As such, it makes no sense to make a large value of K, as the additional clusters have no added importance.