CS273A Homework 6

Due Friday, December 6th, 2024

Instructions

This homework (and many subsequent ones) will involve data analysis and reporting on methods and results using Python code. You will submit a **single PDF file** that contains everything to Gradescope. This includes any text you wish to include to describe your results, the complete code snippets of how you attempted each problem, any figures that were generated, and scans of any work on paper that you wish to include. It is important that you include enough detail that we know how you solved the problem, since otherwise we will be unable to grade it.

Your homeworks will be given to you as Jupyter notebooks containing the problem descriptions and some template code that will help you get started. You are encouraged to use these starter Jupyter notebooks to complete your assignment and to write your report. This will help you not only ensure that all of the code for the solutions is included, but also will provide an easy way to export your results to a PDF file (for example, doing *print preview* and *printing to pdf*). I recommend liberal use of Markdown cells to create headers for each problem and sub-problem, explaining your implementation/answers, and including any mathematical equations. For parts of the homework you do on paper, scan it in such that it is legible (there are a number of free Android/iOS scanning apps, if you do not have access to a scanner), and include it as an image in the Jupyter notebook.

If you have any questions/concerns about using Jupyter notebooks, ask us on EdD. If you decide not to use Jupyter notebooks, but go with Microsoft Word or Latex to create your PDF file, make sure that all of the answers can be generated from the code snippets included in the document.

Summary of Assignment: 100 total points

- Problem 1: Clustering Iris Data (55 points)
  - Problem 1.1: Data Points (5 points)
  - Problem 1.2: K-Means Clustering (15 points)
  - Problem 1.3: K-Means++ Initialization (10 points)
  - Problem 1.4: Selecting a Clustering (5 points)
  - Problem 1.5: Agglomerative Clustering (15 points)
  - Problem 1.5: Analysis (5 points)
- Problem 2: Dimensionality Reduction (40 points)
  - Problem 2.1: Preprocessing (5 points)
  - Problem 2.2: Eigendecomposition (10 points)
  - Problem 2.3: Reconstruction (10 points)
  - Problem 2.4: Visualizing (5 points)
  - Problem 2.5: Nonlinear & Implicit Embeddings (10 points)
- Statement of Collaboration (5 points)

Before we get started, let's import some libraries that you will make use of in this assignment. Make sure that you run the code cell below in order to import these libraries.

Important: In the code block below, we set `seed=1234`. This is to ensure your code has reproducible results and is important for grading. Do not change this. If you are not using the provided Jupyter notebook, make sure to also set the random seed as below.

Important: Do not change any codes we give you below, except for those waiting for you to complete. This is to ensure your code has reproducible results and is important for grading.

```python
import numpy as np
import matplotlib.pyplot as plt

import requests                                      # reading data
from io import StringIO

import warnings
warnings.filterwarnings('ignore')

from sklearn.datasets import load_iris
from sklearn.cluster import KMeans, AgglomerativeClustering
from sklearn.inspection import DecisionBoundaryDisplay

plt.set_cmap('nipy_spectral');

import scipy.linalg

# Fix the random seed for reproducibility
# !! Important !! : do not change this
seed = 1234
np.random.seed(seed)
```

```
<Figure size 640x480 with 0 Axes>
```

Problem 1: Clustering

In this problem you will experiment with two clustering algorithms implemented in `scikit-learn`: k-means and agglomerative clustering.

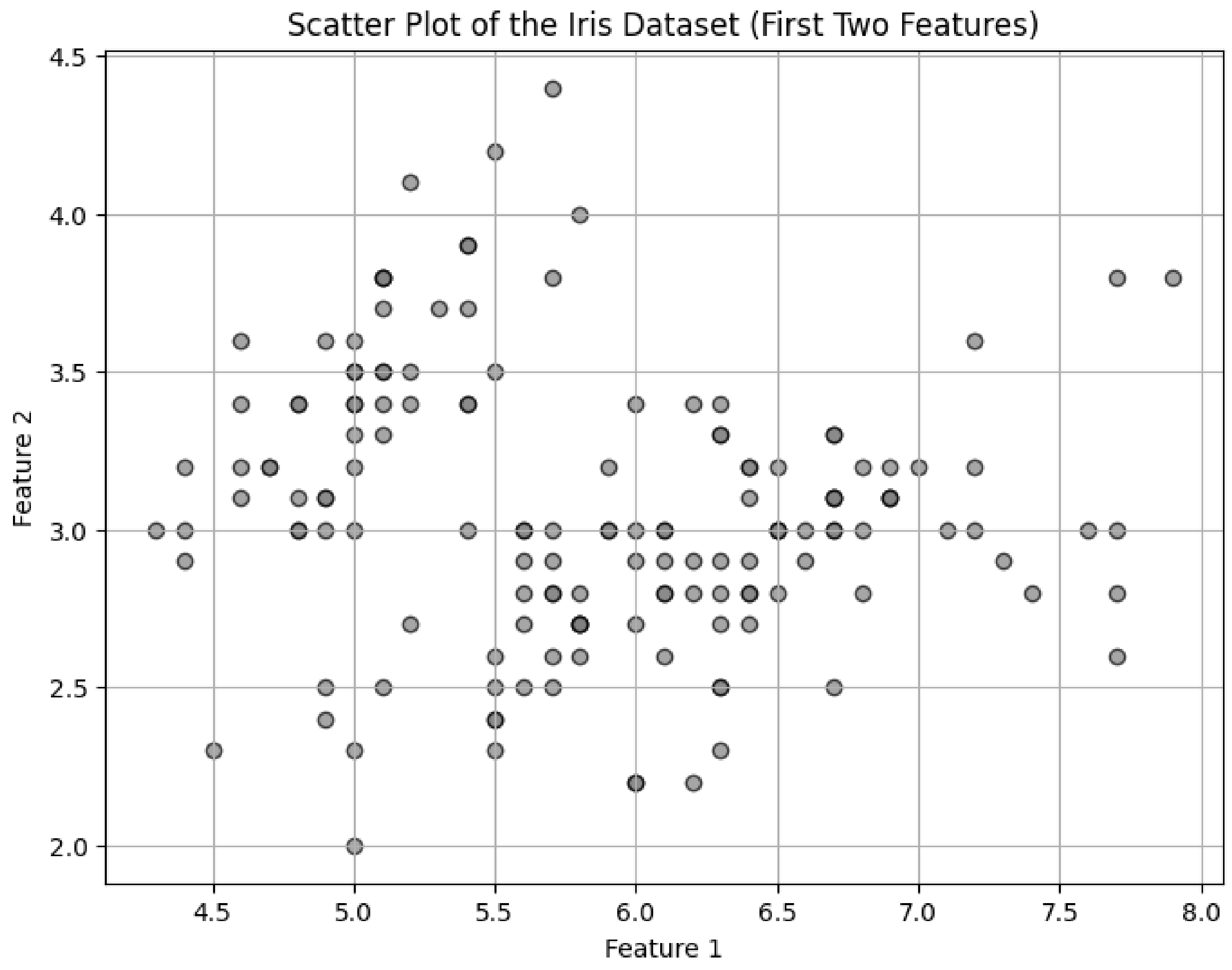
Let's also load in some data that we will use for the tests in Problem 1. Here, we are using the Iris dataset, where we're only using the first two features. Although you typically would split your data into a training set and a testing set, we won't do that here because we are only using this data to illustrate clustering.

```python
# Load the Iris dataset
X, y = load_iris(return_X_y = True)
# Only use the first two features
X = X[:, :2]
```

Problem 1.1: Data Points (10 points):

First, plot the Iris data features X, and see how "clustered" you think it looks. How many clusters do you think there should be for these data?

```python
# Plot the Iris data
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c='gray', edgecolor='k', alpha=0.7)
plt.title("Scatter Plot of the Iris Dataset (First Two Features)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.grid(True)
plt.show()
```



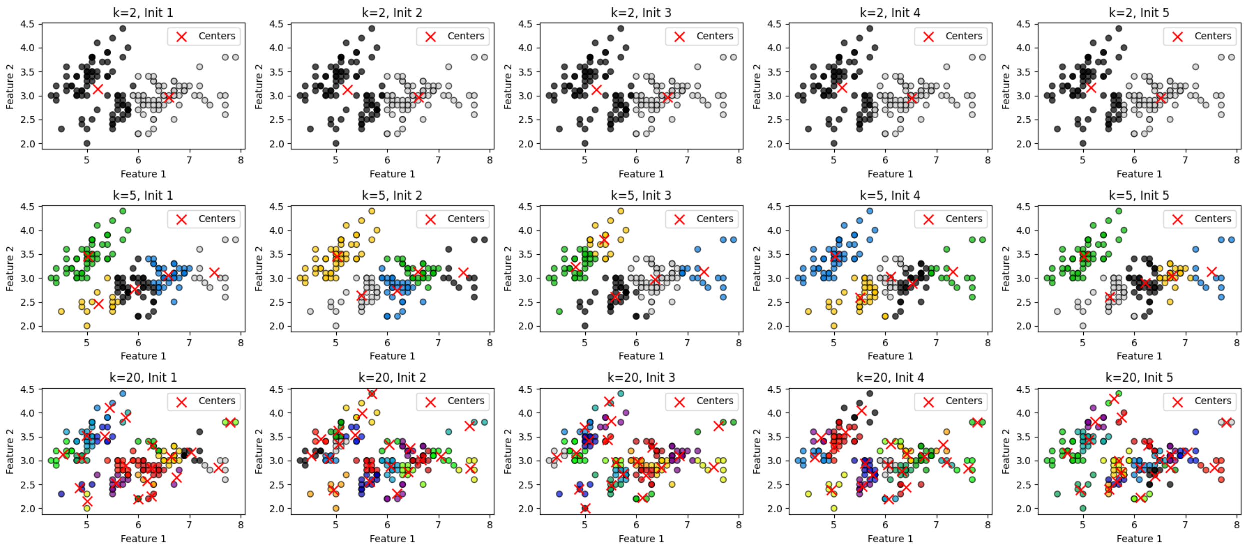
I think there should be roughly 2 clusters for this data

Problem 1.2: K-Means Clustering (15 points):

Run k-means clustering on the data X, for several choices of k: $k \in \{2, 5, 20\}$. Use the basic `random` initialization. *Manually* fit at least 5 different initializations followed by the k-means algorithm (i.e., run the function `.fit(X)` at least 5 times with different random seeds), and for each one, check and compare their cluster quality (either visually by plotting, or by comparing the total distortion via `score`).

Note: it is not usually helpful to compare the cluster labels assigned to the data, since the identity of the each cluster (its index, $0 \ldots k-1$) is not important -- only which data have been assigned to the same cluster.

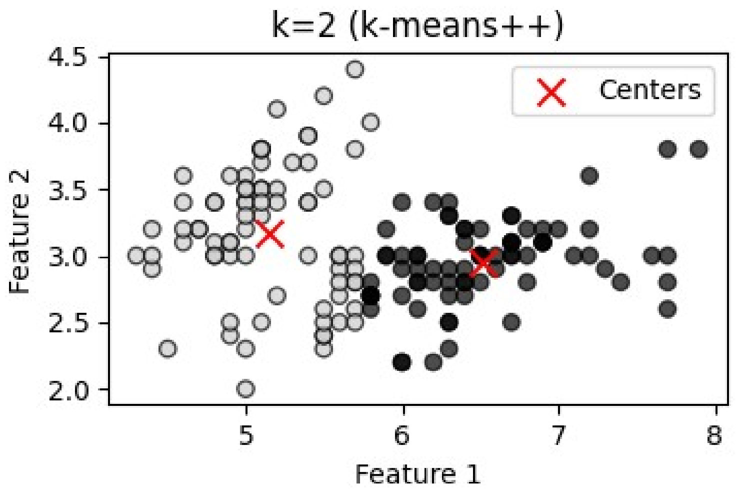
```
fig, ax = plt.subplots(3,5, figsize=(18,8))

for j,k in enumerate([2,5,20]):
    for i in range(5):
        random_state = seed * k + i

        clust = KMeans(n_clusters=k, init='random',
random_state=random_state, n_init=1)
        clust.fit(X)

        labels = clust.predict(X)

        # Plot the clusters
        ax[j, i].scatter(X[:, 0], X[:, 1], c=labels, edgecolor='k',
alpha=0.7)
        ax[j, i].scatter(clust.cluster_centers_[:, 0],
clust.cluster_centers_[:, 1],
                         c='red', marker='x', s=100, label='Centers')
        ax[j, i].set_title(f"k={k}, Init {i+1}")
        ax[j, i].set_xlabel("Feature 1")
        ax[j, i].set_ylabel("Feature 2")
        ax[j, i].legend()

plt.tight_layout()
plt.show()
```

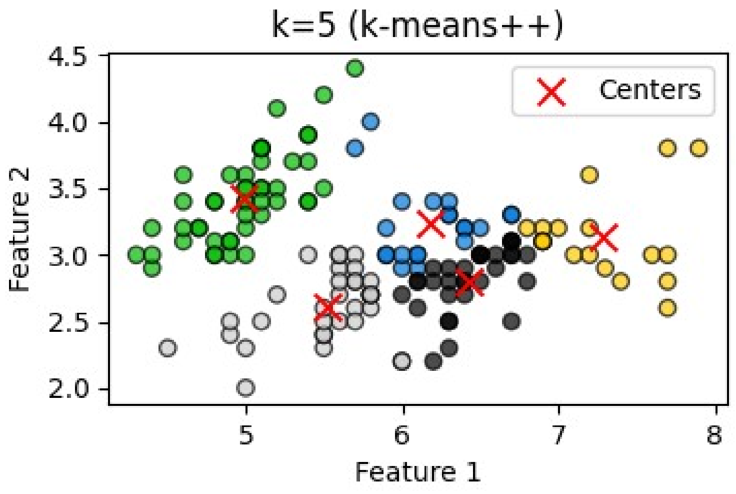


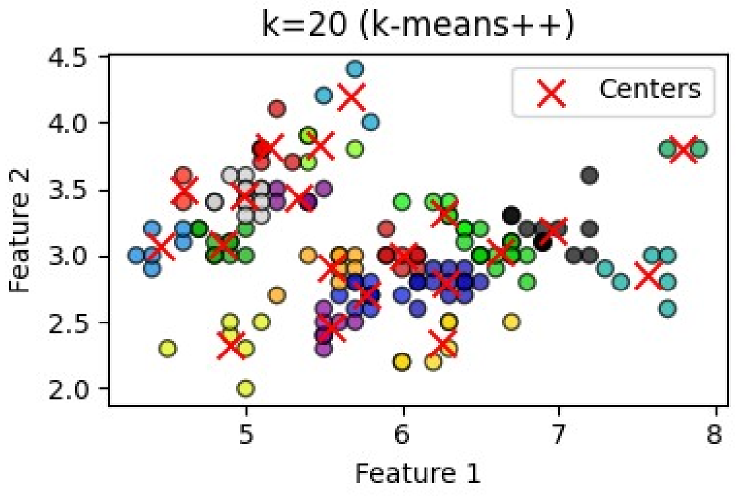
Problem 1.3: K-Means++ Initialization (10 points):

Run a single initialization and fit using the `k++` initialization technique. Compare the resulting clustering to those in the previous problem, both visually and by score.

```python
fig, ax = plt.subplots(3,1, figsize=(4,8))

distortions = []

for j,k in enumerate([2,5,20]):
    for i in range(1):
        # again, use random_state = seed*k + i
        random_state = seed*k + i

        kmeans = KMeans(n_clusters=k, init='k-means++', 
random_state=random_state, n_init=1)
        kmeans.fit(X)

        distortions.append((k , kmeans.inertia_))

        # Predict cluster labels
        labels = kmeans.predict(X)

        # Plot the clusters
        ax[j].scatter(X[:, 0], X[:, 1], c=labels, edgecolor='k', 
alpha=0.7)
        ax[j].scatter(kmeans.cluster_centers_[:, 0], 
kmeans.cluster_centers_[:, 1],
                      c='red', marker='x', s=100, label='Centers')
        ax[j].set_title(f"k={k} (k-means++)")
        ax[j].set_xlabel("Feature 1")
        ax[j].set_ylabel("Feature 2")
        ax[j].legend()

plt.tight_layout()
plt.show()

print("Distortion scores using k-means++:")
for k, distortion in distortions:
    print(f"k={k}: Distortion = {distortion:.2f}")
```







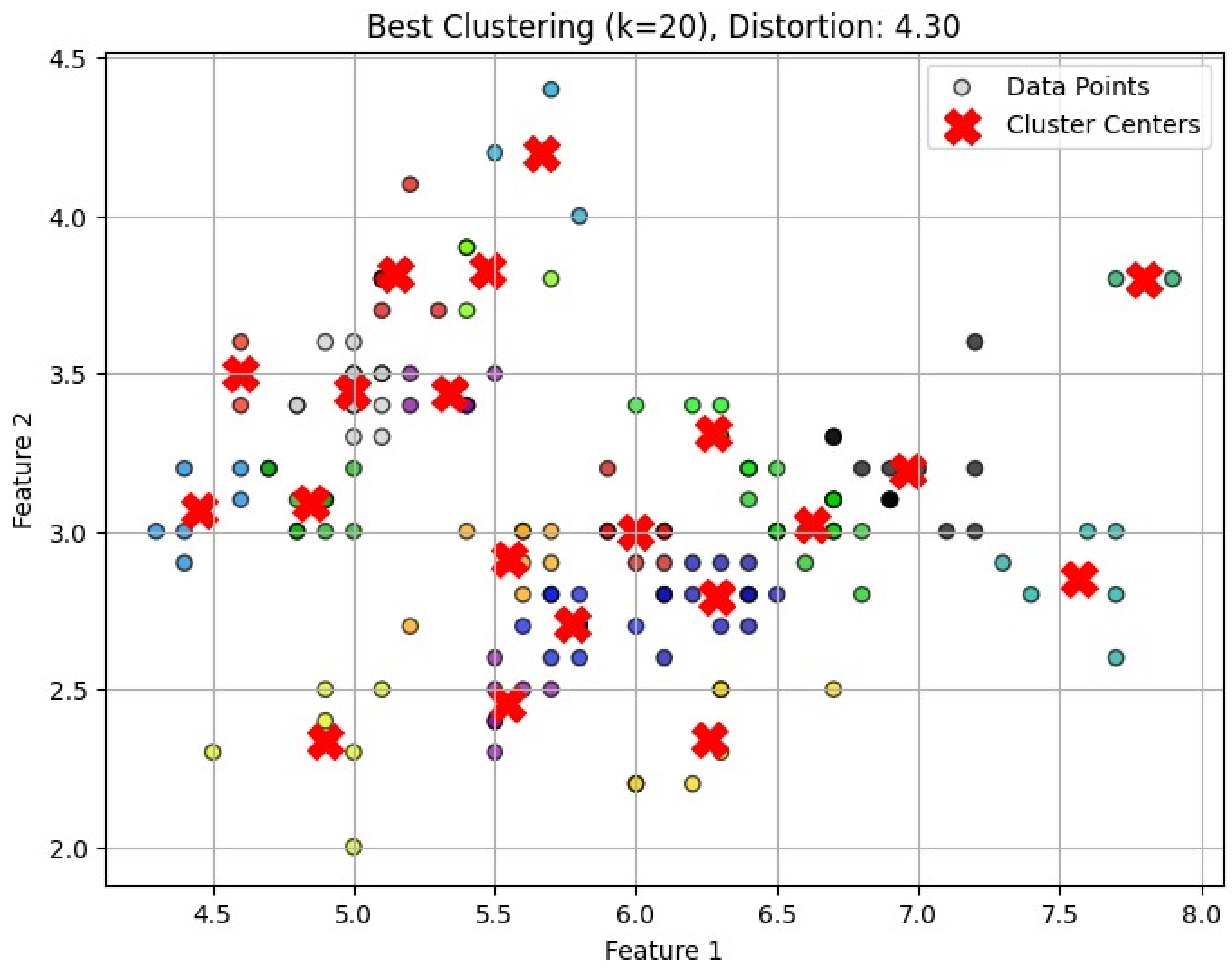
```
Distortion scores using k-means++:
k=2: Distortion = 58.45
k=5: Distortion = 26.06
k=20: Distortion = 4.30
```

Problem 1.4: Selecting a Clustering (5 points):

Select the best clustering (by `score`) and display the data (`scatter`) colored by their cluster membership, along with the cluster centers (as `X` markers). (You can get the closest assigned cluster via `predict`.)

```
best_k, best_distortion = min(distortions, key=lambda x: x[1])
print(f"Best clustering: k={best_k}")
print(f"Distortion score: {best_distortion:.2f}")

# Re-fit the best clustering to retrieve labels and centers
random_state = seed * best_k
best_kmeans = KMeans(n_clusters=best_k, init='k-means++',
random_state=random_state, n_init=1)
best_kmeans.fit(X)
best_labels = best_kmeans.predict(X)

plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=best_labels, edgecolor='k', alpha=0.7,
label='Data Points')
plt.scatter(best_kmeans.cluster_centers_[:, 0],
best_kmeans.cluster_centers_[:, 1],
            c='red', marker='X', s=200, label='Cluster Centers')
plt.title(f"Best Clustering (k={best_k}), Distortion:
{best_distortion:.2f}")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.grid(True)
plt.show()
```

```
Best clustering: k=20
Distortion score: 4.30
```



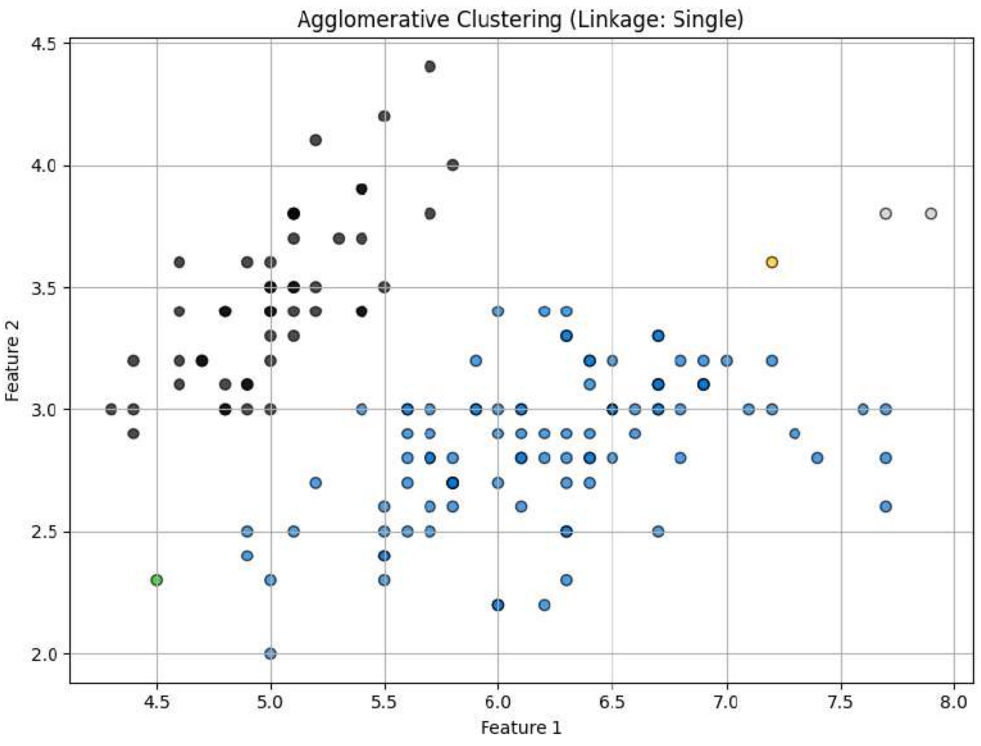
As a note, clustering that can be extended to out-of-sample points, such as k-means, can provide a useful construction of additional features for downstream supervised learning. The `transform` function in the k-means class uses distance to the various clusters as a feature transform, which can replace or augment the original features for a learner.

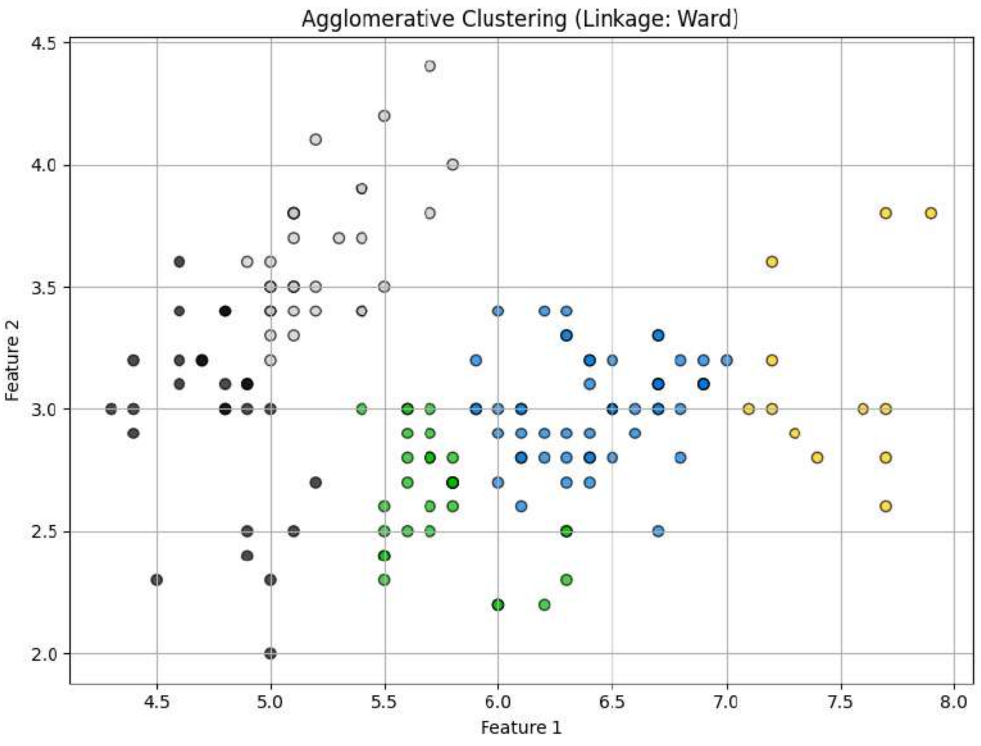
Problem 1.5: Agglomerative Clustering (15 points)

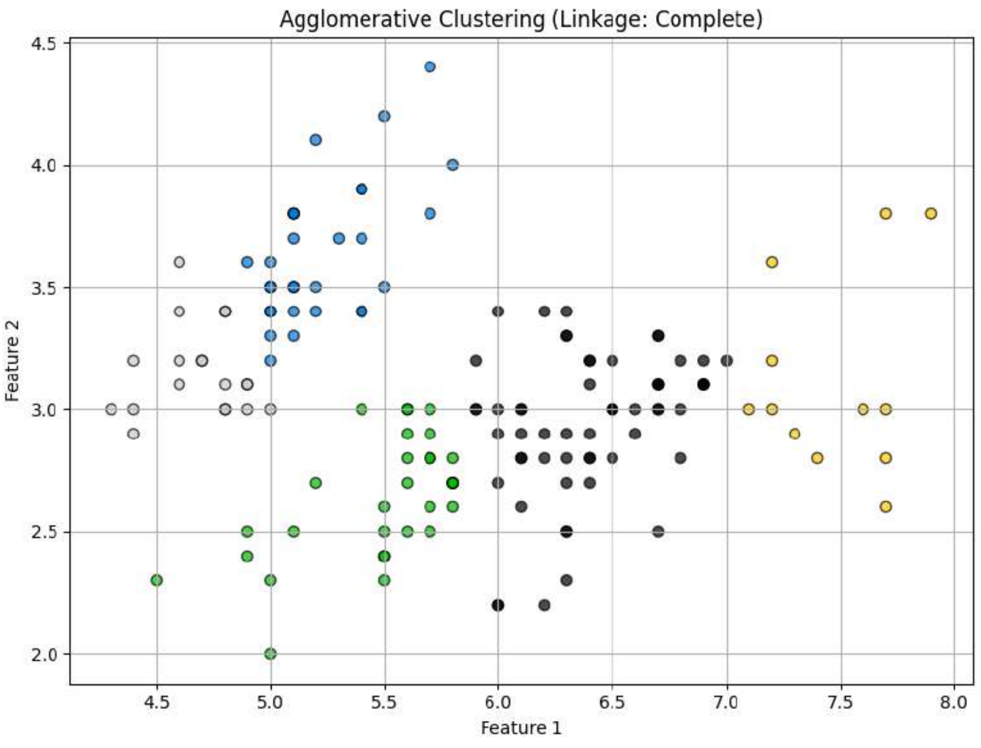
Now use heirarchical aggolomerative clustering to find groupings of the data into 5 clusters (the middle value from the k-means exercise), under different definitions of the "cluster distance": `single` linkage (nearest pair of points), `ward` (mean distance, distance between the means of the clusters), and `complete` linkage (furthest pair of points). Use the usual Euclidean distance as the dissimilarity metric (`metric = 'euclidean'`).

Note that, unlike k-means, the agglomerative clustering procedure is not easily applied to out-of-sample points, meaning that, given a new location x, it is not always clear which cluster it should belong to. (In k-means, we can simply select the nearest cluster center.) In `sklearn`, this is reflected in the fact that there is no `predict` function (to apply the learned clustering to out-of-sample data points); you can access the cluster assignments of the data used during clustering as `labels_`, or call `fit_predict` (which fits and provides the prediction on the data used for fitting).

```python
from sklearn.cluster import AgglomerativeClustering

fig, ax = plt.subplots(3, 1, figsize=(8, 18))

# Define the linkage methods
linkages = ['single', 'ward', 'complete']

# Loop over each linkage method
for i, linkage in enumerate(linkages):
    # Initialize Agglomerative Clustering
    agglom = AgglomerativeClustering(n_clusters=5, linkage=linkage,
metric='euclidean')

    # Fit and predict cluster labels
    labels = agglom.fit_predict(X)

    # Plot the clusters
    ax[i].scatter(X[:, 0], X[:, 1], c=labels, edgecolor='k',
alpha=0.7)
    ax[i].set_title(f"Agglomerative Clustering (Linkage:
{linkage.capitalize()})")
    ax[i].set_xlabel("Feature 1")
    ax[i].set_ylabel("Feature 2")
    ax[i].grid(True)

plt.tight_layout()
plt.show()
```


Agglomerative Clustering (Linkage: Single)
Feature 1
Feature 2


Agglomerative Clustering (Linkage: Ward)
Feature 1
Feature 2


Agglomerative Clustering (Linkage: Complete)
Feature 1
Feature 2

Problem 1.6: Analysis (5 points)

Compare the results of the three clusterings you obtained via agglomerative clustering to the clusterings you obtained from k-means. Do any seem better, or worse, and why?

Why do we not run agglomerative clustering multiple times, as we did with k-means?

Agglomerative clustering (especially Ward linkage) produces similar compact clusters to k-means, but single linkage often creates elongated, poorly separated clusters, making it worse. Complete linkage clusters are better-separated but less compact than k-means. We don't run agglomerative clustering multiple times because it's deterministic, with no random initialization like k-means, so repeated runs give the same result.

Problem 2: Dimensionality Reduction

In this problem, we will use the MNIST dataset to do a little exploration of PCA representations for smoothing or compressing high-dimensional data. First, let's load MNIST:

```
# Load the features and labels for the MNIST dataset
# This might take a minute to download the images.
from sklearn.datasets import fetch_openml           # common data set
access
X_mnist, y_mnist = fetch_openml('mnist_784', as_frame=False,
return_X_y=True)
p = 28                                               # WxH for mnist
data

# Convert labels to integer data type
y_mnist = y_mnist.astype(int)
```

Let's first just keep the data corresponding to the digits "3":

```
X,y = X_mnist[y_mnist==3], y_mnist[y_mnist==3]
```

There are 7141 threes, sized 28x28 = 784 pixels each.

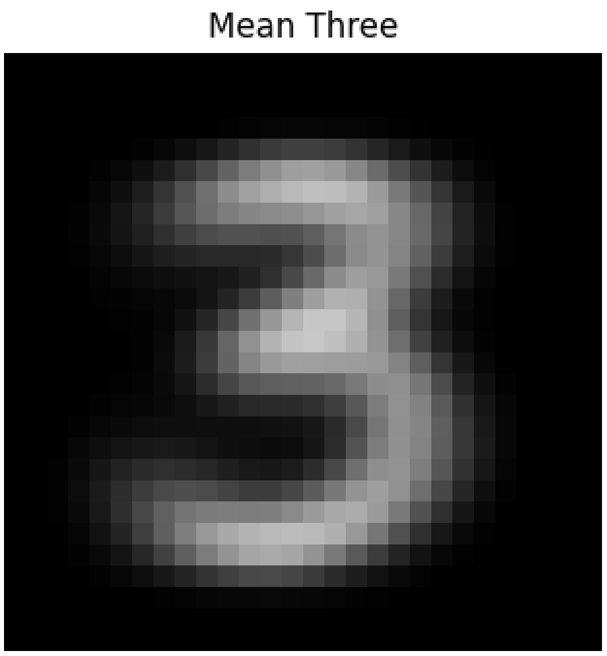
As in our earlier homework, we can display the images using `imshow`; since the images are grayscale only, we use a gray colormap. and specify the range of values in that colormap (`vmin, vmax`) so that the contrast is not adjusted automatically in the display function. For example, we can look at five data points:

```
fig,ax = plt.subplots(1,5, figsize=(12,2));
for a,i in enumerate([1,5,30,50,100]):
    ax[a].imshow(X[i,:].reshape(p,p), cmap="gray", vmin=0,vmax=255);
    ax[a].axis('off')
```

Problem 2.1: Preprocessing (5 points)

First, find the "mean three" (average over the 7000 threes), and display it. Remove this mean vector from the data to obtain a zero-centered data set `X0`:

```python
import numpy as np

mean_three = np.mean(X, axis=0)

plt.figure(figsize=(4, 4))
plt.imshow(mean_three.reshape(p, p), cmap="gray", vmin=0, vmax=255)
plt.title("Mean Three")
plt.axis('off')
plt.show()

# Zero-center the data
X_0 = X - mean_three

print(f"Shape of zero-centered dataset: {X_0.shape}")
```



```
Shape of zero-centered dataset: (7141, 784)
```

Problem 2.2: EigenDecomposition (10 points)

Now, use `scipy.linalg.svd` to compute the singular value decomposition,

$$X0 = U \odot S \odot V^T = Z \odot V^T$$

where U, V^T are unitary (orthogonal) matrices, so that $U \odot U^T = V^T \odot V = I$, and S is diagonal.

(Note that `svd` returns V^T as an output, rather than V.)

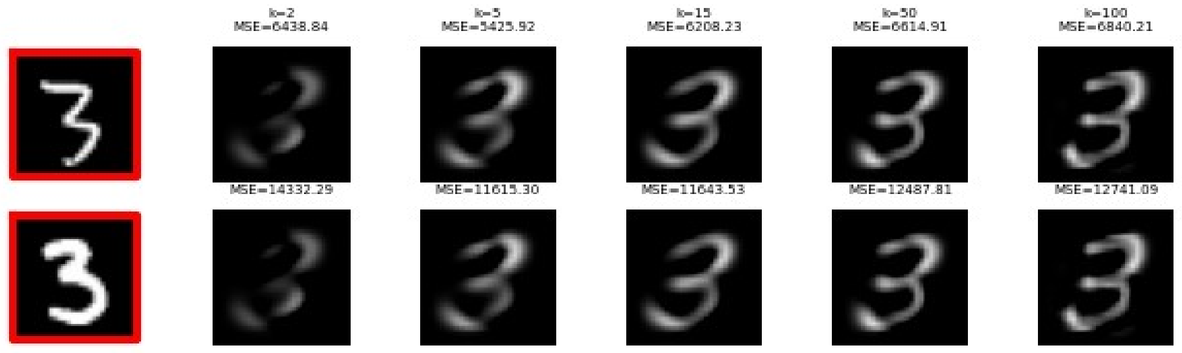
We define $Z = U \odot S$ for convenience.

```python
import scipy.linalg

U, S, VT = scipy.linalg.svd(X_0, full_matrices=False)

Z = U * S[np.newaxis, :]
```

Problem 2.3: Reconstructions (10 points)

Compute the reconstruction of digits 5 and 33 using only the top k eigenvectors, where $k \in \{1, 5, 15, 50, 100\}$. For each reconstruction, what is the distortion (mean squared error between the pixels in the original image $x^{(i)}$ and the reconstructed image $\hat{x}^{(i)}$)?

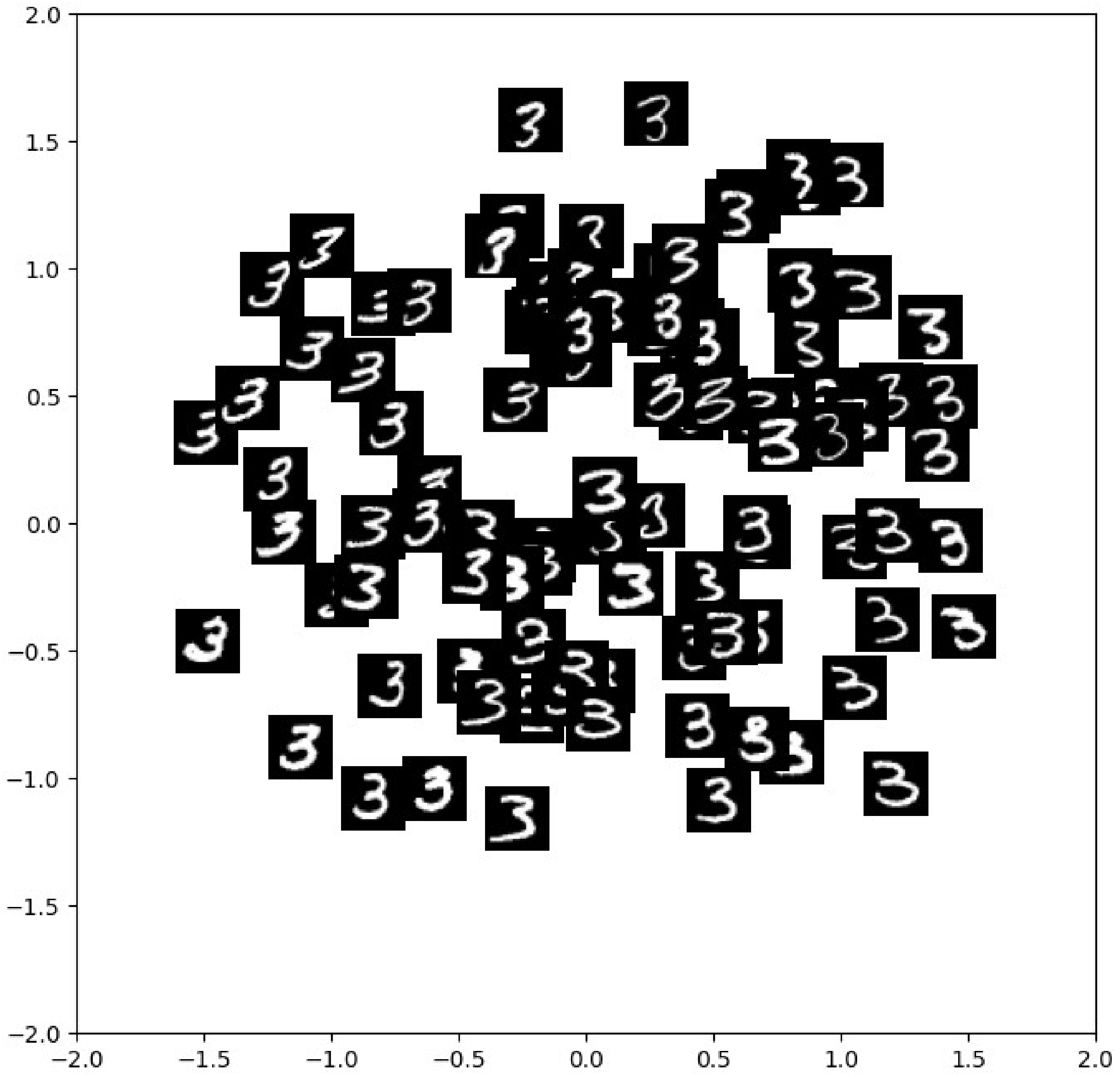
```python
# Reconstruct two digits using a few components
def reconstruct(X_original, U, S, VT, k):
    Uk = U[:1, :k]  # Select the first row of U for the specific image
    Sk = S[:k]
    VTk = VT[:k, :]
    X_reconstructed = Uk @ np.diag(Sk) @ VTk
    distortion = np.mean((X_original - X_reconstructed) ** 2)
    return X_reconstructed, distortion

fig,ax = plt.subplots(2,6, figsize=(8,2))

for ii,i in enumerate([5,33]):
    ax[ii,0].imshow(X[i,:].reshape(p,p), cmap="gray",vmin=0,vmax=255);
    ax[ii,0].axis('off');   # plot &
    ax[ii,0].plot([-1,-1,p,p,-1],[-1,p,p,-1,-1],'r-',lw=3)
# highlight "original" data
    for j,k in enumerate([2,5,15,50,100]):
        # YOUR CODE GOES HERE
        X_reconstructed, distortion = reconstruct(X[i, :].reshape(1, -
1), U, S, VT, k)

        ax[ii, j + 1].imshow(X_reconstructed.reshape(p, p),
cmap="gray", vmin=0, vmax=255)
        ax[ii, j + 1].axis('off')
        ax[ii, j + 1].set_title(f"k={k}\nMSE={distortion:.2f}" ,
fontsize = 5)
```



Problem 2.4: Visualizing the latent space (5 points)

We can create a scatterplot of the digits in their latent position z (the rows of the matrix Z in SVD, or just U if you want to remove the scaling effect of S) to visualize how the data change across the space. **Run** the following (provided) code to do so, and **comment on the axes found** for the layout of images. (For example, the first axis should look like a rotation, or the "slant" of the digit -- why does this make sense?)

```
# Let's plot some of the digits and see what the first two dimensions
look like...
W = Z

np.random.seed(1234)
idx = np.floor( len(X)*np.random.rand(100) ); # pick some data
idx = idx.astype('int')

plt.figure(figsize=(8,8))

scale = 0.08/np.std(W)
for i in idx:
    loc = (scale*W[i,0],scale*W[i,0]+.25,
scale*W[i,1],scale*W[i,1]+.25)
    plt.imshow(np.reshape(X[i,:],(p,p)) , cmap="gray", extent=loc ,
vmin=0,vmax=255);

plt.axis( (-2,2,-2,2) );
```



comments:

First Axis: Represents a visual feature like the "slant" or "rotation" of the digit "3."

Second Axis: Captures another variation, such as the roundness or openness of the loops in the digit "3."

Problem 2.5 Nonlinear & Implicit Embeddings (10 points)

While PCA is fast and relatively easy to understand, high dimensional data often does not fall nicely within a small linear vector subspace. As discussed in class, one option is to train a nonlinear autoencoder, which replaces the linear projection operations in PCA with a nonlinear function. But, another option is to simply optimize directly over the latent locations $z^{(i)}$ of each data point i, which defines an implicit embedding (i.e., we do not know the function that maps x

to z, just its values at the data points). To do this, we must define a "loss" which accounts for whether the locations $z^{(i)}$ "match" the original data $x^{(i)}$. We'll compare the PCA embedding with a particular implicit embedding called TSNE.

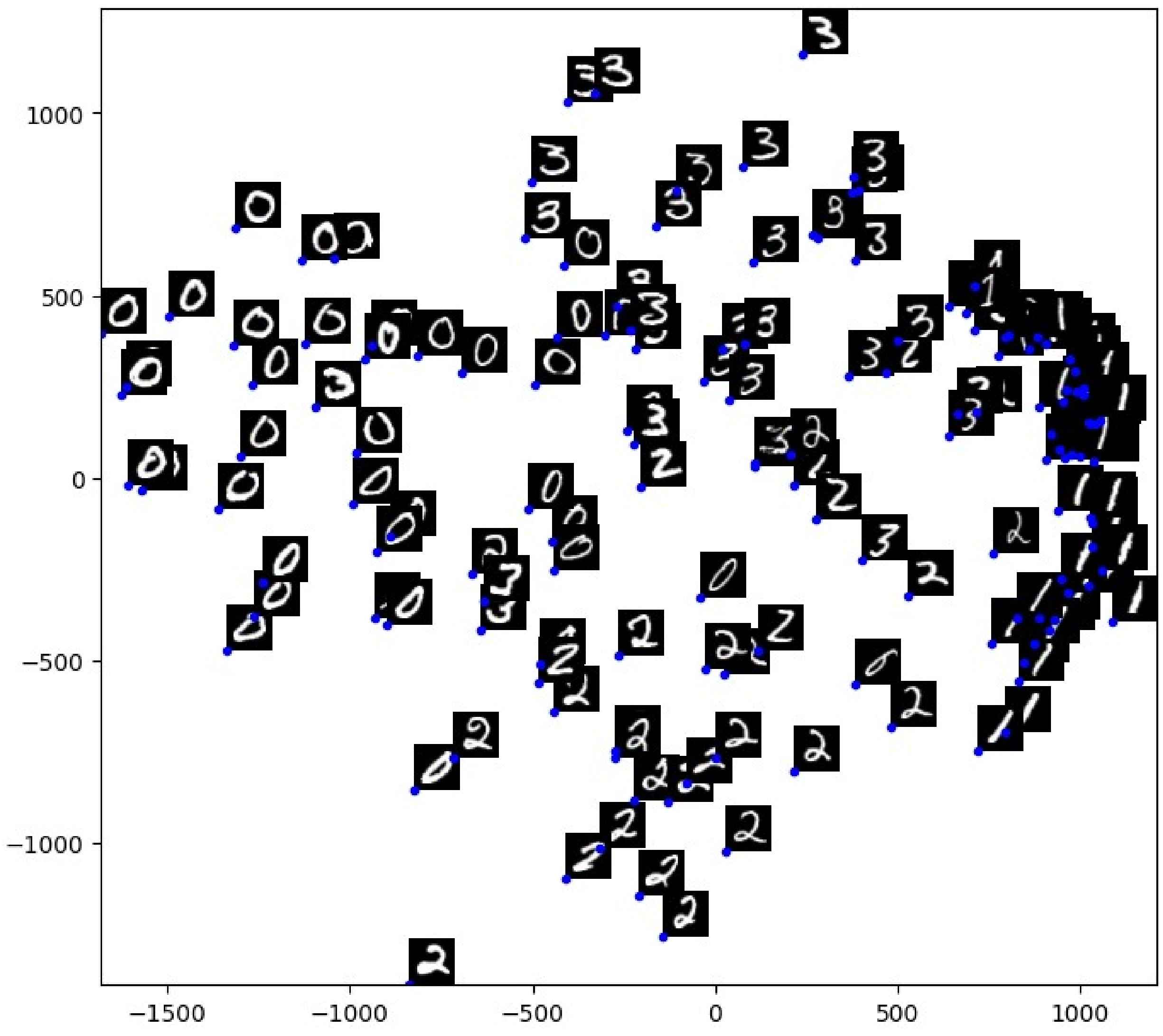
Note: this problem doesn't really require you to solve anything; just perform the embeddings and interpret their results.

```
# First, let's grab a few more digits of MNIST:
X,y = X_mnist[y_mnist<=3], y_mnist[y_mnist<=3]

np.random.seed(1234)
idx = np.floor( len(X)*np.random.rand(150) ); # pick some data
idx = idx.astype('int')
```

PCA representation of 0..3:

First, use PCA to reduce the images to two dimensions. For convenience we'll just use `sklearn` for this here:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(X)
Z = pca.transform(X)

plt.figure(figsize=(8,8))

scale = 500   # need to choose an appropriate scale, compare to the
axes & "extent" used for the image display

for i in idx:
    loc = (Z[i,0],Z[i,0]+scale*.25, Z[i,1],Z[i,1]+scale*.25)
    plt.imshow(np.reshape(X[i,:],(p,p)) , cmap="gray", extent=loc ,
vmin=0,vmax=255);
    plt.plot(loc[0],loc[2],'b.')
    #plt.axis( (-1,1,-1,1) )

plt.show()
```



TSNE nonlinear embedding of 0..3:

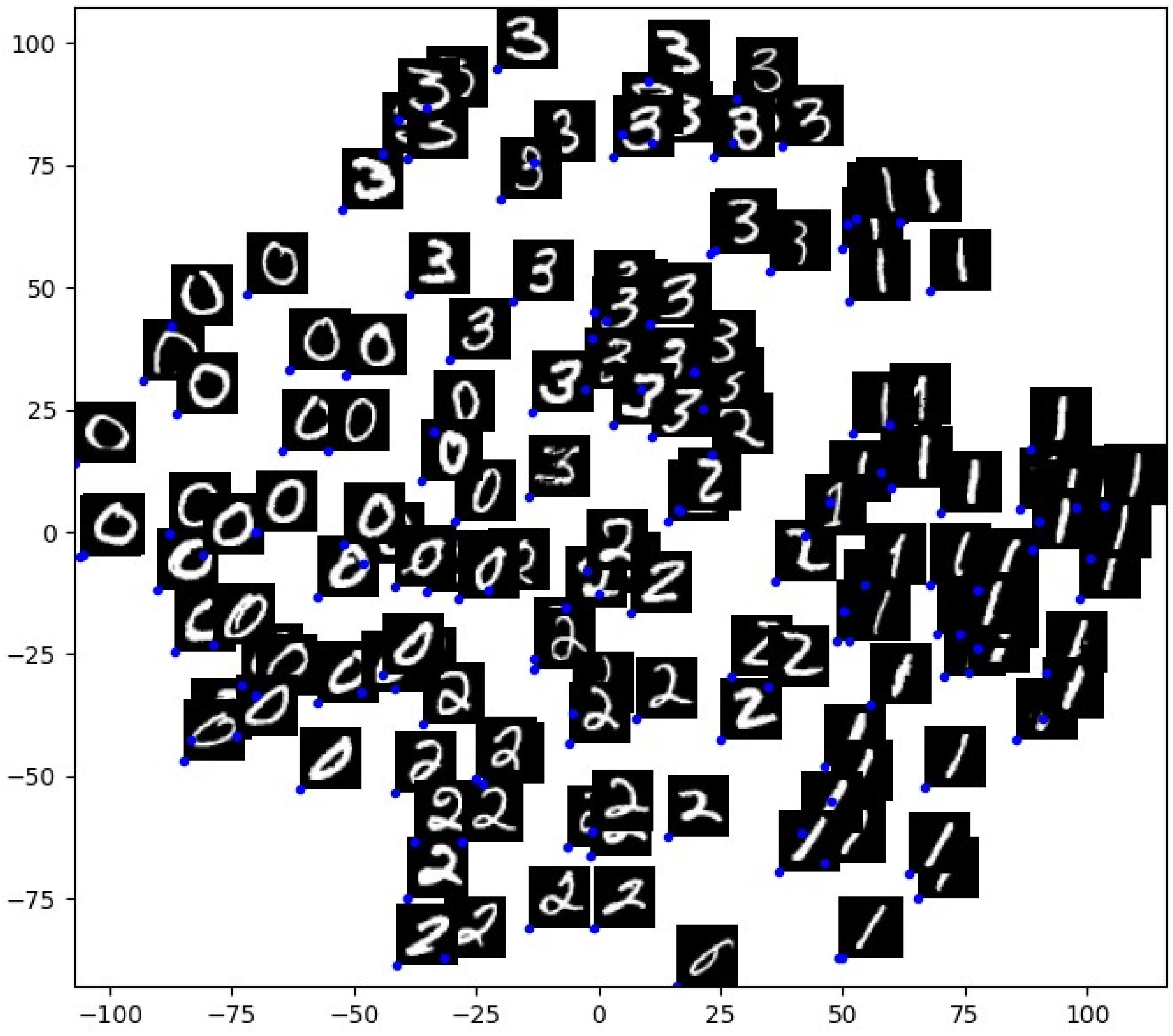
Next, use TSNE to embed the same data. This will be **much** slower, but instead of simply finding a linear subspace, will place the data at positions z to preserve their relative "similarity". (You can see the TSNE page or the course notes for more precise details.) The `perplexity` parameter effectively determines the neighborhood size of this similarity comparison; it has a significant effect on the results and can be hard to set automatically.

```
# Computationally intensive, "nonlinear" embedding...
from sklearn.manifold import TSNE

# Note: this can be a bit slow!
embedding = TSNE(n_components=2, learning_rate='auto', init='pca',
verbose=1, perplexity=10)
Z = embedding.fit_transform(X)
```

```
[t-SNE] Computing 31 nearest neighbors...
[t-SNE] Indexed 28911 samples in 0.010s...
[t-SNE] Computed neighbors for 28911 samples in 62.892s...
[t-SNE] Computed conditional probabilities for sample 1000 / 28911
[t-SNE] Computed conditional probabilities for sample 2000 / 28911
[t-SNE] Computed conditional probabilities for sample 3000 / 28911
[t-SNE] Computed conditional probabilities for sample 4000 / 28911
[t-SNE] Computed conditional probabilities for sample 5000 / 28911
[t-SNE] Computed conditional probabilities for sample 6000 / 28911
[t-SNE] Computed conditional probabilities for sample 7000 / 28911
[t-SNE] Computed conditional probabilities for sample 8000 / 28911
[t-SNE] Computed conditional probabilities for sample 9000 / 28911
[t-SNE] Computed conditional probabilities for sample 10000 / 28911
[t-SNE] Computed conditional probabilities for sample 11000 / 28911
[t-SNE] Computed conditional probabilities for sample 12000 / 28911
[t-SNE] Computed conditional probabilities for sample 13000 / 28911
[t-SNE] Computed conditional probabilities for sample 14000 / 28911
[t-SNE] Computed conditional probabilities for sample 15000 / 28911
[t-SNE] Computed conditional probabilities for sample 16000 / 28911
[t-SNE] Computed conditional probabilities for sample 17000 / 28911
[t-SNE] Computed conditional probabilities for sample 18000 / 28911
[t-SNE] Computed conditional probabilities for sample 19000 / 28911
[t-SNE] Computed conditional probabilities for sample 20000 / 28911
[t-SNE] Computed conditional probabilities for sample 21000 / 28911
[t-SNE] Computed conditional probabilities for sample 22000 / 28911
[t-SNE] Computed conditional probabilities for sample 23000 / 28911
[t-SNE] Computed conditional probabilities for sample 24000 / 28911
[t-SNE] Computed conditional probabilities for sample 25000 / 28911
[t-SNE] Computed conditional probabilities for sample 26000 / 28911
[t-SNE] Computed conditional probabilities for sample 27000 / 28911
[t-SNE] Computed conditional probabilities for sample 28000 / 28911
[t-SNE] Computed conditional probabilities for sample 28911 / 28911
[t-SNE] Mean sigma: 290.438975
[t-SNE] KL divergence after 250 iterations with early exaggeration:
100.193932
[t-SNE] KL divergence after 1000 iterations: 2.605052
```

```
np.random.seed(1234)
idx = np.floor( len(X)*np.random.rand(150) ); # pick some data
idx = idx.astype('int')

plt.figure(figsize=(8,8))

scale = 50.   # need to choose an appropriate scale, compare to the
axes & "extent" used for the image display

for i in idx:
    loc = (Z[i,0],Z[i,0]+scale*.25, Z[i,1],Z[i,1]+scale*.25)
    plt.imshow(np.reshape(X[i,:],(p,p)) , cmap="gray", extent=loc ,
vmin=0,vmax=255);
```

```
    plt.plot(loc[0],loc[2],'b.')
    #plt.axis( (-1,1,-1,1) )

plt.show()
```



Compare the two embeddings (PCA vs TSNE). How are they similar? How do they differ? Why might this be the case? (The differences are even more pronounced if you keep all 10 digits; try it if you like.)

How Are PCA and t-SNE Similar?

Both PCA and t-SNE aim to reduce the high-dimensional MNIST data to a more manageable 2D space for visualization. They both reveal clusters of digits, showing that the data has some inherent structure. For instance, in both plots, digits like "0" and "3" form recognizable groups, meaning the embeddings are capturing meaningful patterns.

How Do They Differ?

- **Cluster Separation:** PCA creates overlapping clusters, especially for digits with similar shapes, like "2" and "3." In contrast, t-SNE does a much better job of separating these clusters, making it easier to see the distinctions between digits.

- **Axis Meaning:** In PCA, the axes have a clear interpretation. They represent the principal components, which are combinations of pixel intensities that capture the most variance in the data. In t-SNE, however, the axes don't have a specific meaning.

PCA is designed to maximize variance in a straight-line projection. It's great for datasets where the important information lies along linear dimensions, but it struggles with the complex, curved patterns that often exist in real-world data. t-SNE doesn’t assume linearity; instead, it prioritizes preserving how similar data points are to each other, even if that means distorting the larger structure.

The differences between PCA and t-SNE should become clearer when we include all 10 digits.

Why does TSNE only have a `fit_transform` function, while PCA has separate `fit` and `transform` functions?

t-SNE only has fit_transform because it's a non-parametric method that directly optimizes the positions of data points in lower dimensions based on pairwise similarities, without learning a reusable mapping. It can't be applied to new, unseen data. In contrast, PCA is a parametric method that computes a fixed set of principal components during the fit step, and these components can later be used to transform new data with the transform method, making PCA reusable across different datasets.

Statement of Collaboration (5 points)

It is **mandatory** to include a Statement of Collaboration in each submission, with respect to the guidelines below. Include the names of everyone involved in the discussions (especially in-person ones), and what was discussed. If you did not collaborate with anyone, you should write something like "I completed this assignment without any collaboration."

All students are required to follow the academic honesty guidelines posted on the course website. For programming assignments, in particular, I encourage the students to organize (perhaps using EdD) to discuss the task descriptions, requirements, bugs in my code, and the relevant technical content before they start working on it. However, you should not discuss the specific solutions, and, as a guiding principle, you are not allowed to take anything written or drawn away from these discussions (i.e. no photographs of the blackboard, written notes, referring to EdD, etc.). Especially after you have started working on the assignment, try to restrict the discussion to EdD as much as possible, so that there is no doubt as to the extent of your collaboration.

I looked at sklearn documentation and discussed with a friend who is also a PhD student in UCI CS.