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.

In [12]:
```
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.

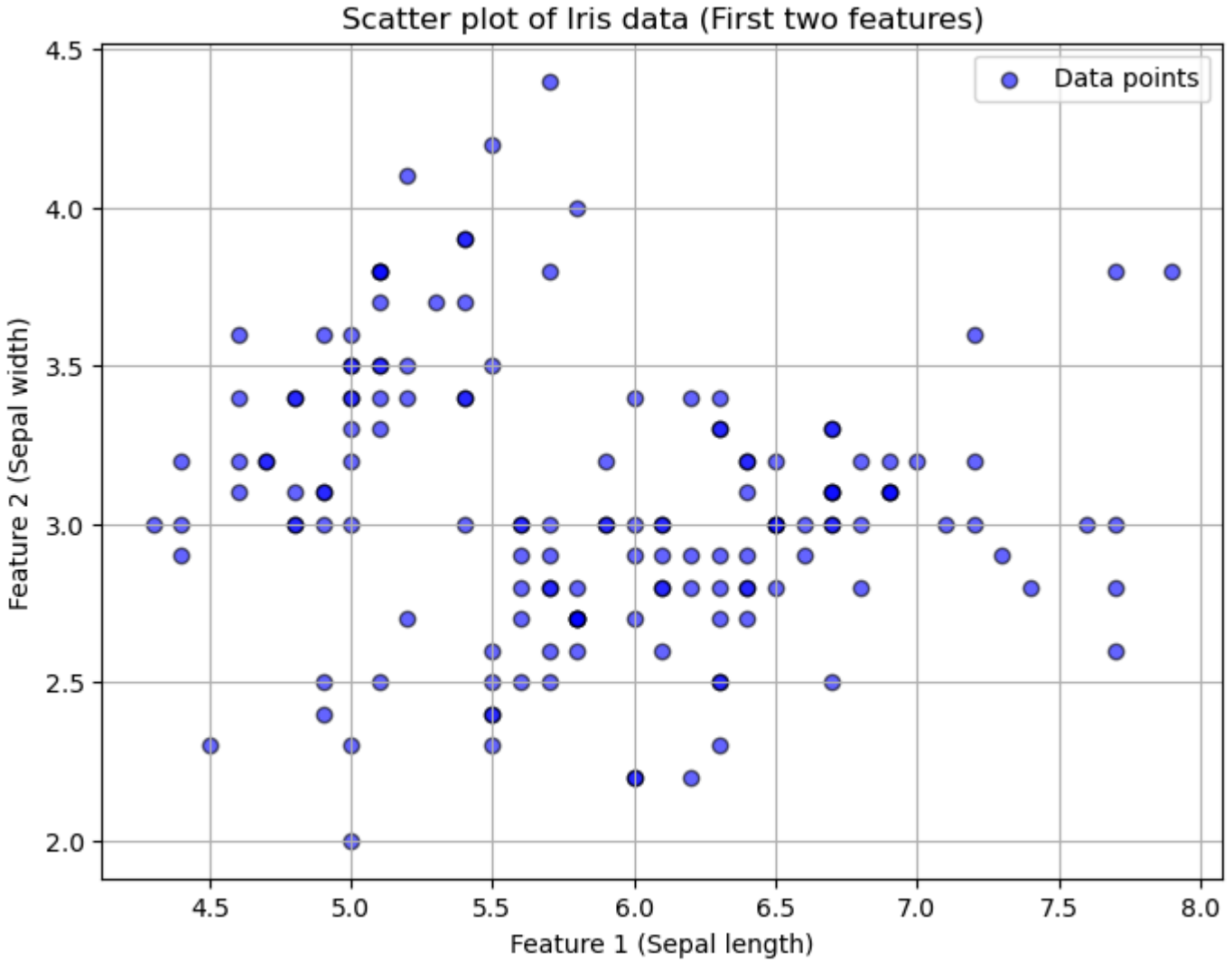
In [7]:
```
# Load the Iris dataset
X, y = load_iris(return_X_y = True)
# Only use the first two features
X = X[:, :2]
```

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?

In [10]:
```
# Scatter plot of the data
plt.figure(figsize=(8, 6))
```

```
plt.scatter(X[:, 0], X[:, 1], c='blue', alpha=0.6, edgecolor='k', label='Data points')
plt.xlabel('Feature 1 (Sepal length)')
plt.ylabel('Feature 2 (Sepal width)')
plt.title('Scatter plot of Iris data (First two features)')
plt.legend()
plt.grid(True)
plt.show()
```



DISCUSS: The data appears somewhat clustered, with identifiable groups or regions where points are denser. However, the clusters are not perfectly distinct, and there is some overlap between groups. Visually, the plot suggests around three clusters.

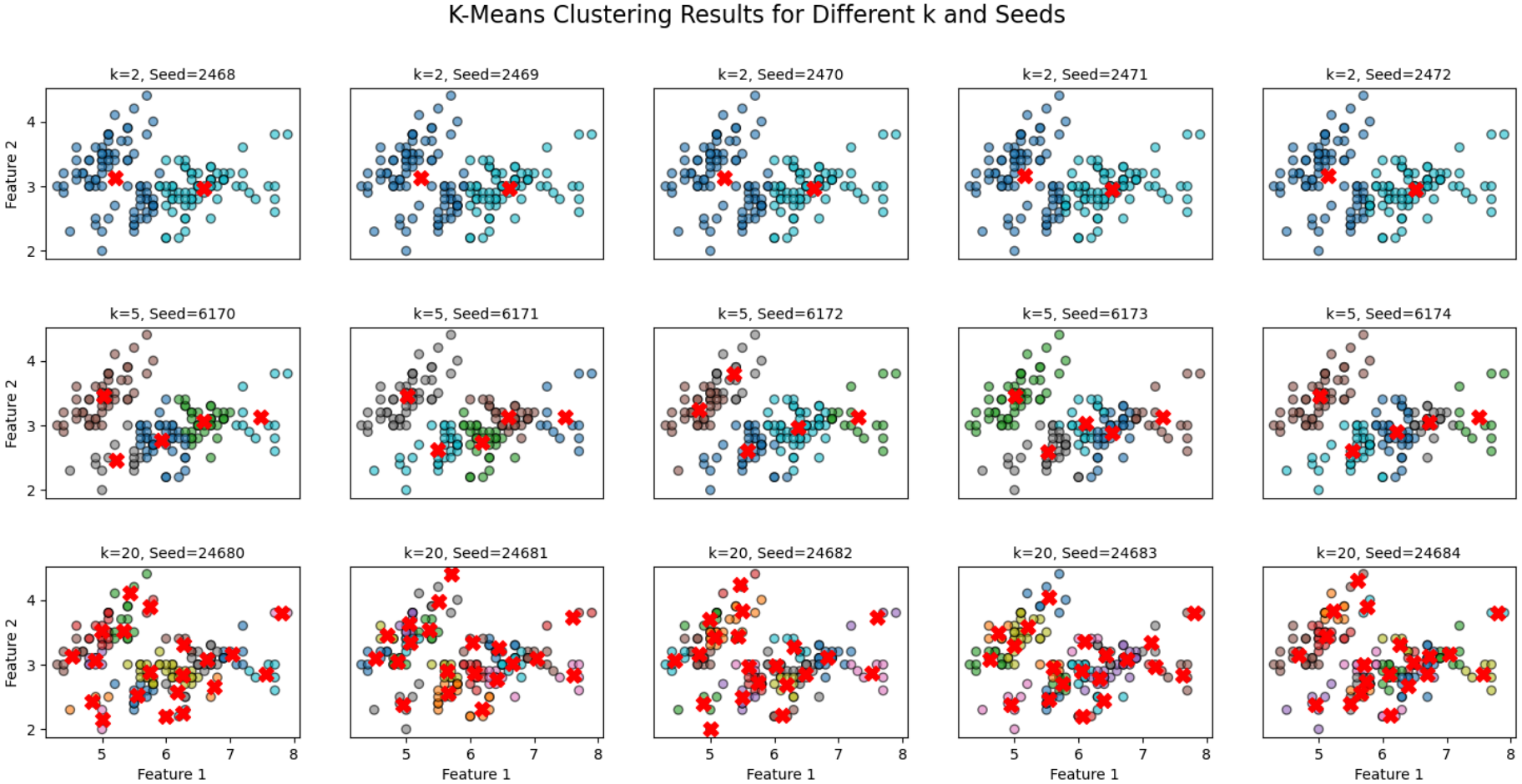
Problem 1.2: K-Means Clustering (15 points):

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.

In [39]:
```
fig, ax = plt.subplots(3,5, figsize=(18,8))
results = []


for j, k in enumerate([2, 5, 20]):
    for i in range(5):
        # Use random_state = seed * k + i
        random_state = k * seed + i
        clust = KMeans(n_clusters=k, init='random', n_init=1, random_state=random_state)
        clust.fit(X)

        # Record results
        score = clust.score(X)
        results.append({'k': k, 'seed': random_state, 'score': score})

        # Scatter plot of clusters
        ax[j, i].scatter(X[:, 0], X[:, 1], c=clust.labels_, cmap='tab10', alpha=0.6, edgecolor='k')
        ax[j, i].scatter(clust.cluster_centers_[:, 0], clust.cluster_centers_[:, 1], c='red', s=100, marker='X', la
        ax[j, i].set_title(f'k={k}, Seed={random_state}', fontsize=10)

        # Set axis labels selectively
        if j == 2:  # Last row: show x-label
            ax[j, i].set_xlabel('Feature 1')
        else:
```

```
            ax[j, i].set_xticks([])  # Hide x-axis ticks

        if i == 0:  # First column: show y-label
            ax[j, i].set_ylabel('Feature 2')
        else:
            ax[j, i].set_yticks([])  # Hide y-axis ticks

# Adjust spacing between subplots
plt.subplots_adjust(wspace=0.2, hspace=0.4)
plt.suptitle('K-Means Clustering Results for Different k and Seeds', fontsize=16)
plt.show()

# Display results
for result in results:
    print(f"k={result['k']}, seed={result['seed']}, score={result['score']}")
```


K-Means Clustering Results for Different k and Seeds
k=2, Seed=2468
k=2, Seed=2469
k=2, Seed=2470
k=2, Seed=2471
k=2, Seed=2472
k=5, Seed=6170
k=5, Seed=6171
k=5, Seed=6172
k=5, Seed=6173
k=5, Seed=6174
k=20, Seed=24680
k=20, Seed=24681
k=20, Seed=24682
k=20, Seed=24683
k=20, Seed=24684
Feature 1
Feature 2

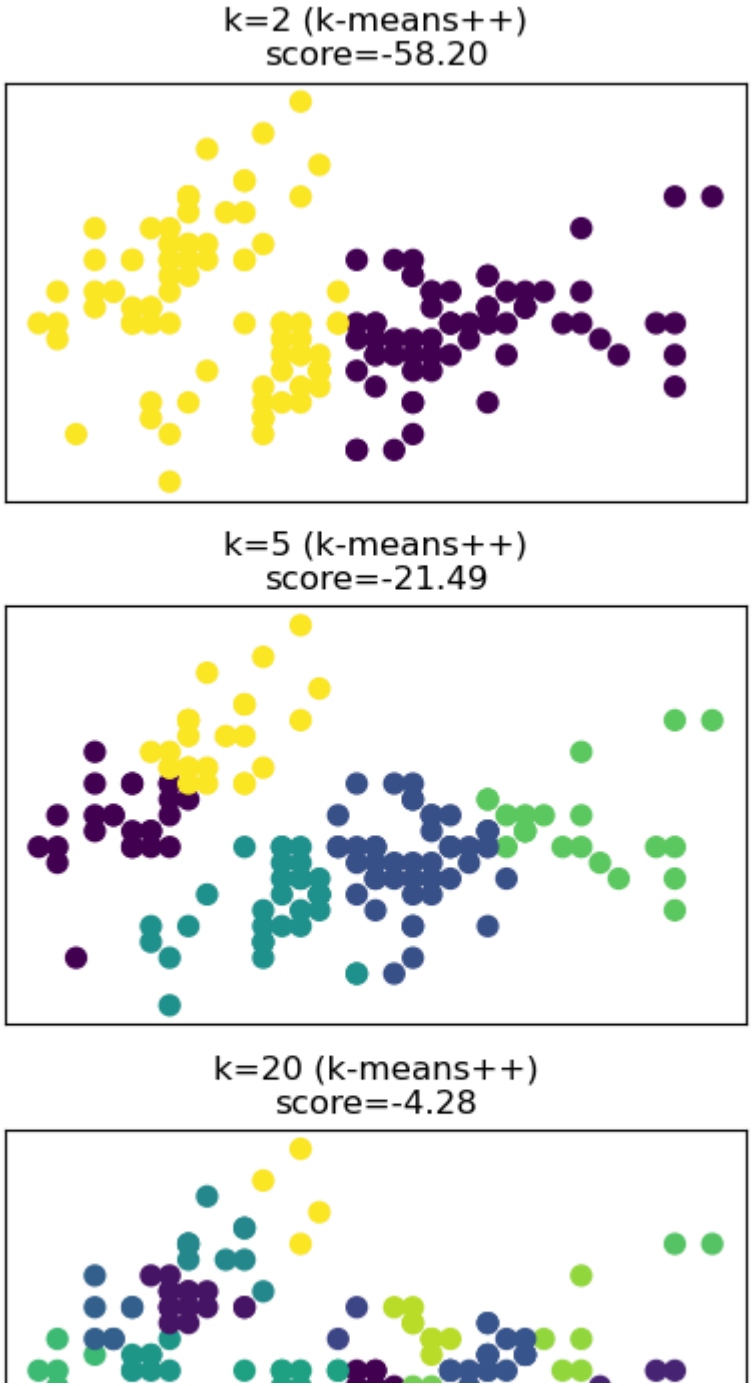
```
k=2, seed=2468, inertia=-58.21499731615674
k=2, seed=2469, inertia=-58.20409278906672
k=2, seed=2470, inertia=-58.20409278906672
k=2, seed=2471, inertia=-58.447592460881935
k=2, seed=2472, inertia=-58.447592460881935
k=5, seed=6170, inertia=-23.862816326530613
k=5, seed=6171, inertia=-24.6004166761254
k=5, seed=6172, inertia=-21.61376186838372
k=5, seed=6173, inertia=-25.951866810360507
k=5, seed=6174, inertia=-24.96755368550582
k=20, seed=24680, inertia=-5.04511994000152
k=20, seed=24681, inertia=-4.544971001221001
k=20, seed=24682, inertia=-4.5915478981802496
k=20, seed=24683, inertia=-4.698815656565658
k=20, seed=24684, inertia=-5.127700673251758
```

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.

In [43]:
```python
fig, ax = plt.subplots(3,1, figsize=(4,8))
results_kmeans_plus = []
for j, k in enumerate([2, 5, 20]):
    for i in range(1):  # Only run one initialization
        random_state = k * seed + i

        # Run KMeans with k-means++ initialization
        clust = KMeans(n_clusters=k, init='k-means++', n_init=10, random_state=random_state)

        # Fit the model
        clust.fit(X)

        # Get the score (negative inertia)
        score = clust.score(X)

        # Store the result
        results_kmeans_plus.append({'k': k, 'seed': random_state, 'score': score})
```

```
        # Plot the result
        ax[j].scatter(X[:, 0], X[:, 1], c=clust.labels_, cmap='viridis', s=50)
        ax[j].set_title(f'k={k} (k-means++)\nscore={score:.2f}')
        ax[j].set_xticks(())
        ax[j].set_yticks(())

# Display the plot
plt.tight_layout()
plt.show()

# Print results for score comparison
print("\nClustering Quality (Score Comparison) for k-means++:")
for result in results_kmeans_plus:
    print(f"k={result['k']}, seed={result['seed']}, score={result['score']:.2f}")
```


k=2 (k-means++)
score=-58.20
k=5 (k-means++)
score=-21.49
k=20 (k-means++)
score=-4.28

```
Clustering Quality (Score Comparison) for k-means++:
k=2, seed=2468, score=-58.20
k=5, seed=6170, score=-21.49
k=20, seed=24680, score=-4.28
```

DISCUSS: From the figures, we can see that k-means++ produces more coherent and well-separated clusters compared to k-mean, especially for larger k values. Based on the score comparison, k-means++ can achieve a higher score under different k values.

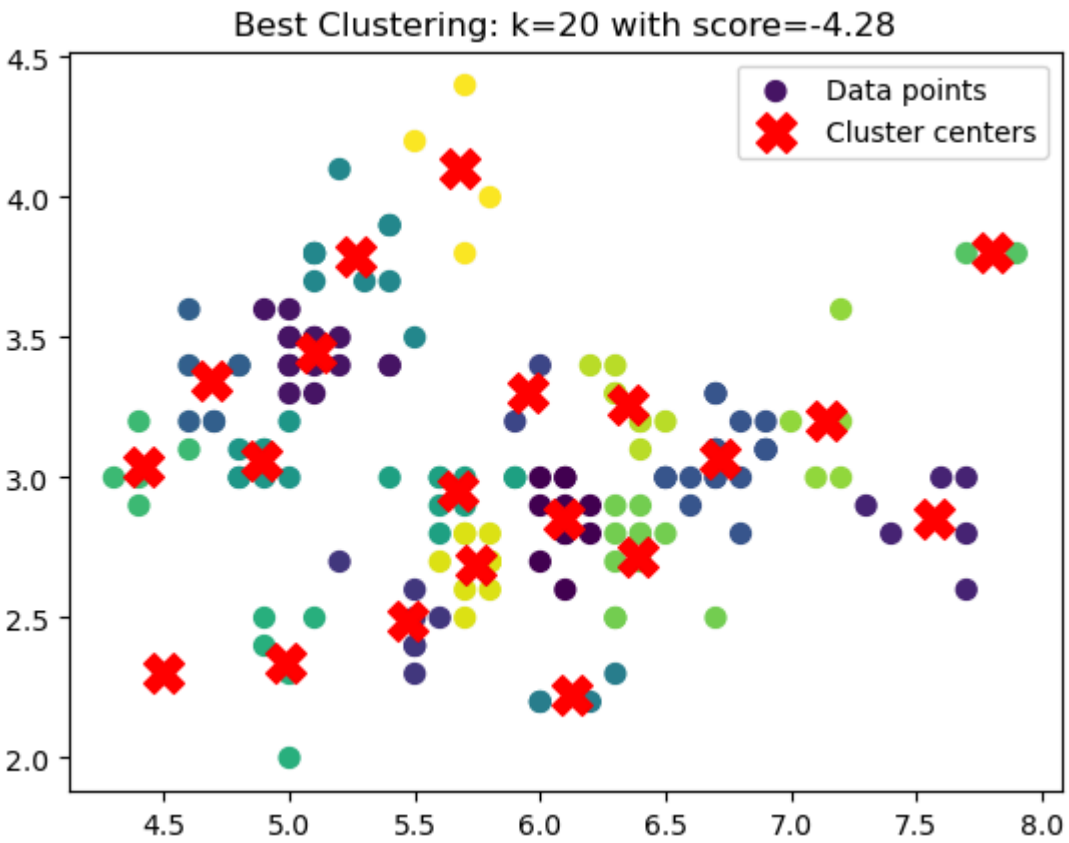
Problem 1.4: Selecting a Clustering (5 points):

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`.)

In [71]:
```python
# Initialize variables to store the best clustering
best_score = float('-inf')  # Initialize to a very low value (since inertia is negative)
best_kmeans = None

# Loop over different values of k and perform clustering with k-means++
for k in [2, 5, 20]:
    for i in range(1):  # Only run one initialization for each k
        random_state = k * seed + i  # Use seed*k + i for reproducibility

        # Run KMeans with k-means++ initialization
        clust = KMeans(n_clusters=k, init='k-means++', n_init=10, random_state=random_state)
        clust.fit(X)

        # Get the score (negative inertia)
        score = clust.score(X)
        print(f'k={k}, seed={random_state}, score={score:.2f}')  # Debug print

        # Update the best clustering if this one is better
        if score > best_score:
            best_score = score
```

```
            best_kmeans = clust

# Check if best_kmeans was updated
if best_kmeans is None:
    print("No valid clustering found.")
else:
    # Get the best clustering labels and cluster centers
    labels = best_kmeans.predict(X)
    centers = best_kmeans.cluster_centers_

    # Plot the results
    plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50, label="Data points")
    plt.scatter(centers[:, 0], centers[:, 1], c='red', marker='X', s=200, label="Cluster centers")
    plt.title(f'Best Clustering: k={best_kmeans.n_clusters} with score={best_score:.2f}')
    plt.legend()
    plt.show()
```

```
k=2, seed=2468, score=-58.20
k=5, seed=6170, score=-21.49
k=20, seed=24680, score=-4.28
```



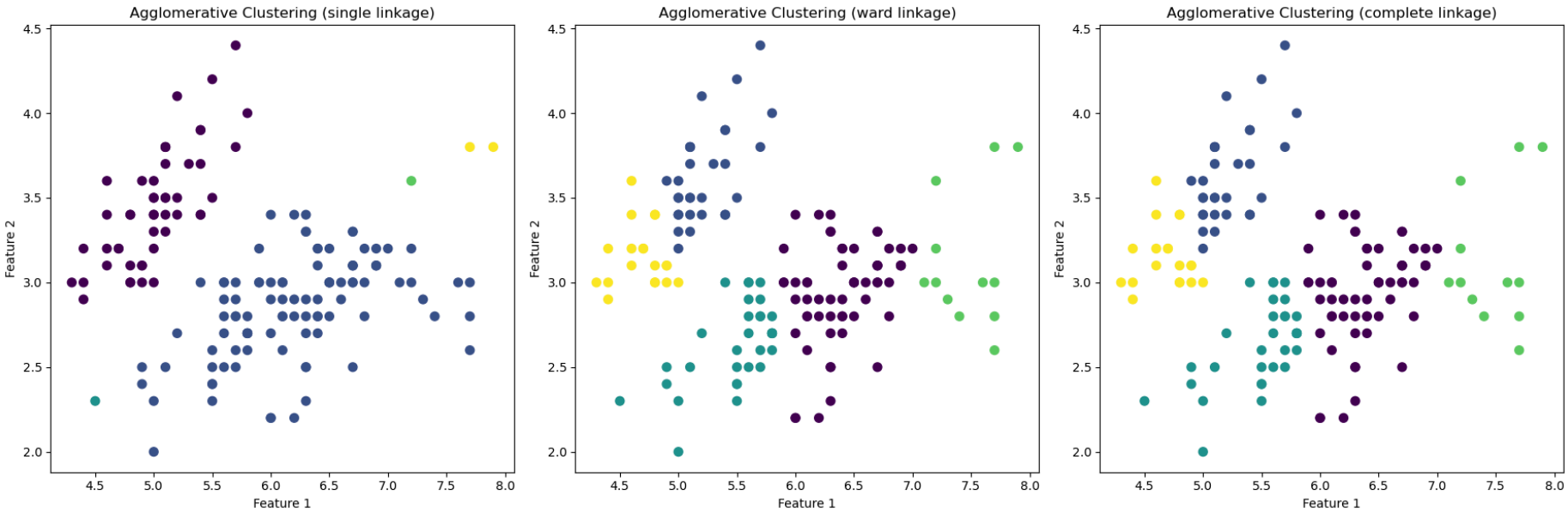
As a note, clustering that can be extended to out-of-sample points, such as k-means, can provide a useful construction of additional features for downstream supervised learning. The `transform` function in the k-means class uses distance to the various clusters as a feature transform, which can replace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

In [69]:
```python
# Linkage methods to compare
linkage_methods = ['single', 'ward', 'complete']

# Create subplots to display the results
fig, ax = plt.subplots(1, 3, figsize=(18, 6))

# Perform clustering with each linkage method
for i, linkage in enumerate(linkage_methods):
    # Perform agglomerative clustering
    cluster = AgglomerativeClustering(n_clusters=5, metric ='euclidean', linkage=linkage)
    labels = cluster.fit_predict(X)  # Get the cluster assignments

    # Plot the results
    ax[i].scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50, label="Data points")
    ax[i].set_title(f'Agglomerative Clustering ({linkage} linkage)')
    ax[i].set_xlabel('Feature 1')
    ax[i].set_ylabel('Feature 2')

# Show the plots
plt.tight_layout()
plt.show()
```



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?

DISCUSS: (1) Agglomerative clustering looks better than k-means. Agglomerative Clustering is a hierarchical clustering method that builds a cluster tree by successively merging the closest clusters. It doesn't assume that clusters are spherical and can adapt to various cluster shapes, making it better at capturing complex, non-convex structures in the data. In contrast, k-means assumes that each cluster is spherical (or at least roughly circular) and defines cluster boundaries based on Euclidean distance from centroids. If the data does not fit this assumption, k-means may produce suboptimal clustering results.

(2) K-means clustering is sensitive to the initial placement of centroids. Agglomerative clustering, on the other hand, doesn't involve centroids or an iterative optimization process like k-means. It works by successively merging clusters based on a linkage criterion, and once the clusters are formed, the assignment will be the same. Because the algorithm is deterministic (given the same data and parameters), there is no need to run it multiple times to ensure a better solution.

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:

In [275…
```
# 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":

In [315…
```
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:

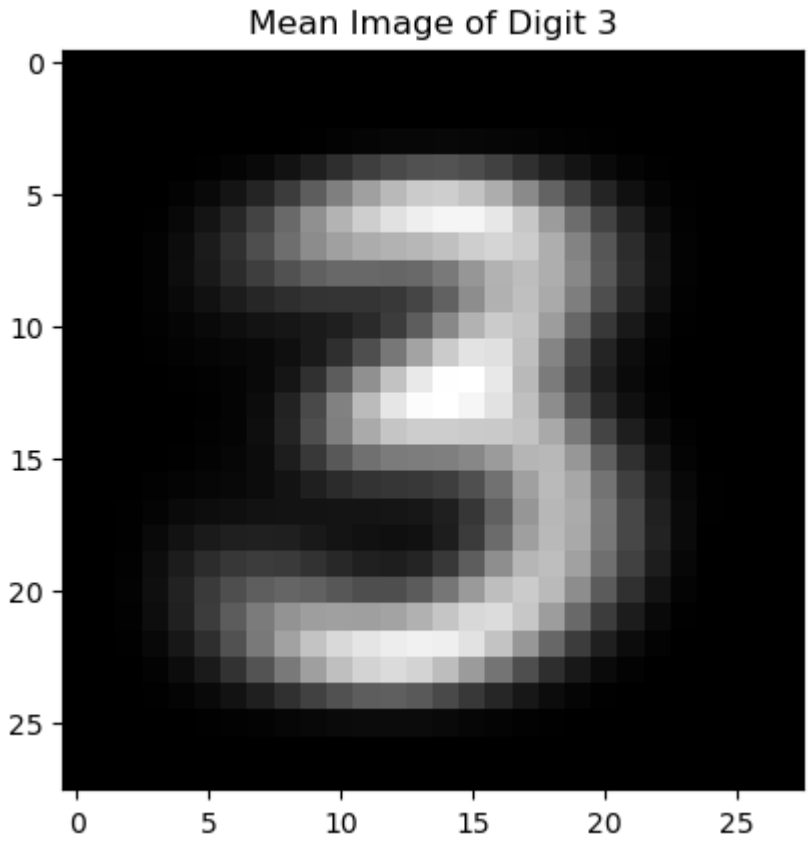
In [279…
```
fig,ax = plt.subplots(1,5, figsize=(12,2));
for a,i in enumerate([1,5,30,50,100]):
    ax[a].imshow(X[i,:].reshape(p,p), cmap="gray", vmin=0,vmax=255);
    ax[a].axis('off')
```

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`:

In [284…

```
mean_three = np.mean(X, axis=0)
mean_image = mean_three.reshape(28, 28)
plt.imshow(mean_image, cmap='gray')
plt.title('Mean Image of Digit 3')
plt.show()

# Subtract the mean vector from each sample to obtain zero-centered data
X0 = X - mean_three
```



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.

In [287…
```
# Perform SVD on the zero-centered data
U, S, Vt = scipy.linalg.svd(X0, full_matrices=False)

# S is returned as a 1D array, so we need to convert it to a diagonal matrix
S_diag = np.diag(S)

# Compute Z = U * S (where S is the diagonal matrix of singular values)
Z = np.dot(U, S_diag)

# Display the shape of the resulting matrices
print(f"Shape of U: {U.shape}")
print(f"Shape of S: {S.shape}")
print(f"Shape of V^T: {Vt.shape}")
print(f"Shape of Z: {Z.shape}")
```

```
Shape of U: (7141, 784)
Shape of S: (784,)
Shape of V^T: (784, 784)
Shape of Z: (7141, 784)
```

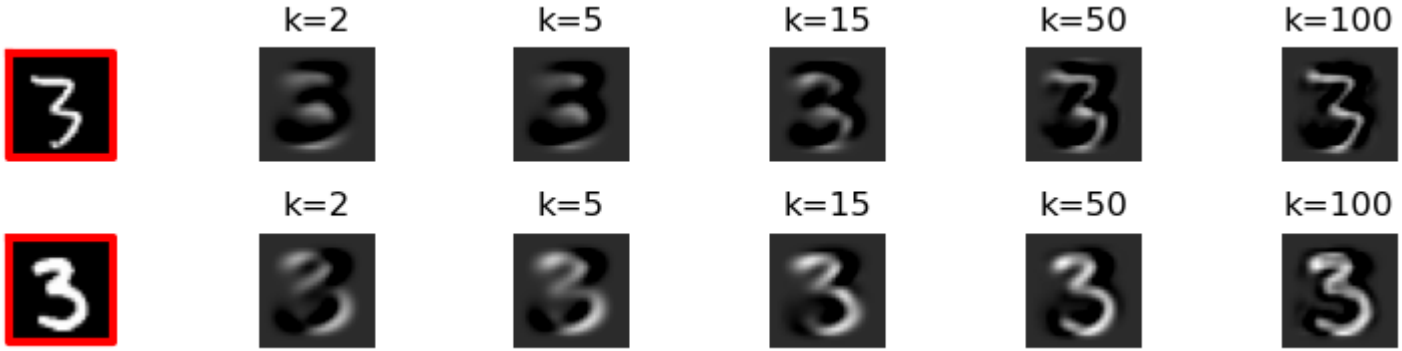
Problem 2.3: Reconstructions (10 points)

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 $\hat{x}^{(i)}$)?

In [305…
```
# Reconstruct two digits using a few components
def reconstruct_image(U, S, Vt, k, mean):
    X_reconstructed = np.dot(U[:, :k], np.dot(np.diag(S[:k]), Vt[:k, :]))
    return X_reconstructed + mean

def mean_squared_error(x, x_reconstructed):
    return np.mean((x - x_reconstructed) ** 2)

mse_values = {}
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
        X0 = X[[i], :]  # Get the current digit (either 5 or 33)
        # Perform SVD on the data and center it
        # mean = X0.mean()   # Mean of the data
        # X0_centered = X0 - mean  # Centered data
        # U, S, Vt = np.linalg.svd(X0_centered, full_matrices=False)  # Perform SVD
        # print(f"Shape of U: {U.shape}")
        # print(f"Shape of S: {S.shape}")
        # print(f"Shape of V^T: {Vt.shape}")
        # Reconstruct the image using the top k eigenvectors
        reconstructed_image = reconstruct_image(U, S, Vt, k, mean)

       # Plot the reconstructed image
        ax[ii, j + 1].imshow(reconstructed_image[i,:].reshape(p,p), cmap="gray",vmin=0,vmax=255);
        ax[ii, j + 1].axis('off')
        ax[ii, j + 1].set_title(f'k={k}')

        mse = mean_squared_error(X0, reconstructed_image[i,:])
        mse_values[(i, k)] = mse

for (i, k), mse in mse_values.items():
    print(f"Digit {i}, k={k}: MSE = {mse:.4f}")

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

```
Digit 5, k=2: MSE = 4128.7197
Digit 5, k=5: MSE = 4440.1347
Digit 5, k=15: MSE = 4081.4185
Digit 5, k=50: MSE = 3538.6072
Digit 5, k=100: MSE = 3157.2023
Digit 33, k=2: MSE = 6431.7362
Digit 33, k=5: MSE = 4570.5712
Digit 33, k=15: MSE = 3881.5327
Digit 33, k=50: MSE = 3336.0254
Digit 33, k=100: MSE = 3064.1212
```

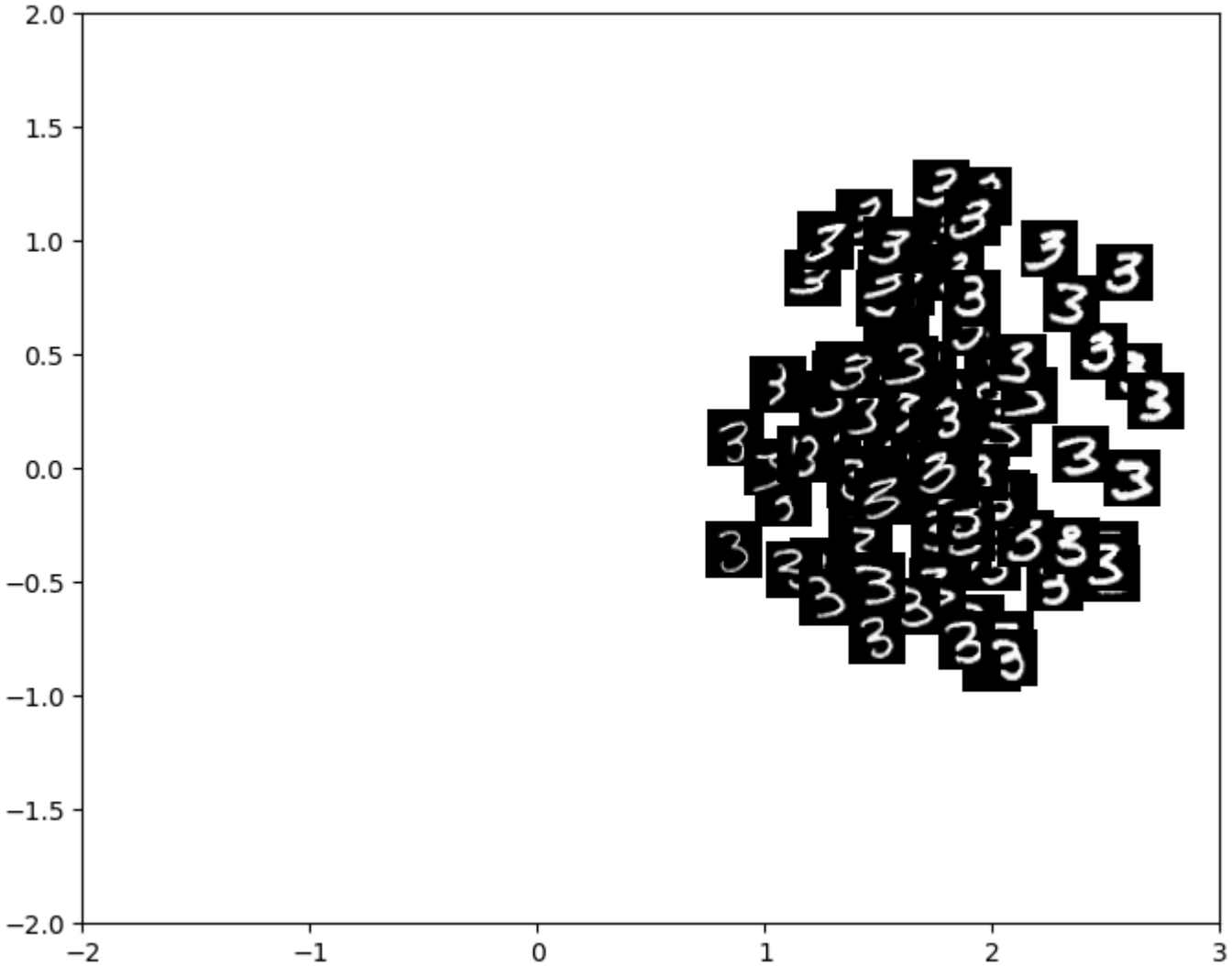


Problem 2.4: Visualizing the latent space (5 points)

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?)

In [338…

```
# Let's plot some of the digits and see what the first two dimensions look like...
np.random.seed(1234)
U, S, Vt = np.linalg.svd(X, full_matrices=False)
# W = Z = U @ S
W = U @ np.diag(S)
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( (-2,3,-2,2) );
```



DISCUSS: From the figure, it can be seen that the first axis (x-axis) is related to the thickness of the font, and as the value of x increases, the font becomes thicker. The second axis (y-axis) is related to the degree of font tilt, with larger y-axis values causing the font to tilt to the right. This is reasonable because these two features are the main variation patterns of fonts. SVD decomposition transforms the original data into a new space, and the first few axes capture the largest changes in the data. Therefore, the axes we see can well explain the main patterns of data changes.

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.

In [105…
```
# First, let's grab a few more digits of MNIST:
X,y = X_mnist, y_mnist

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

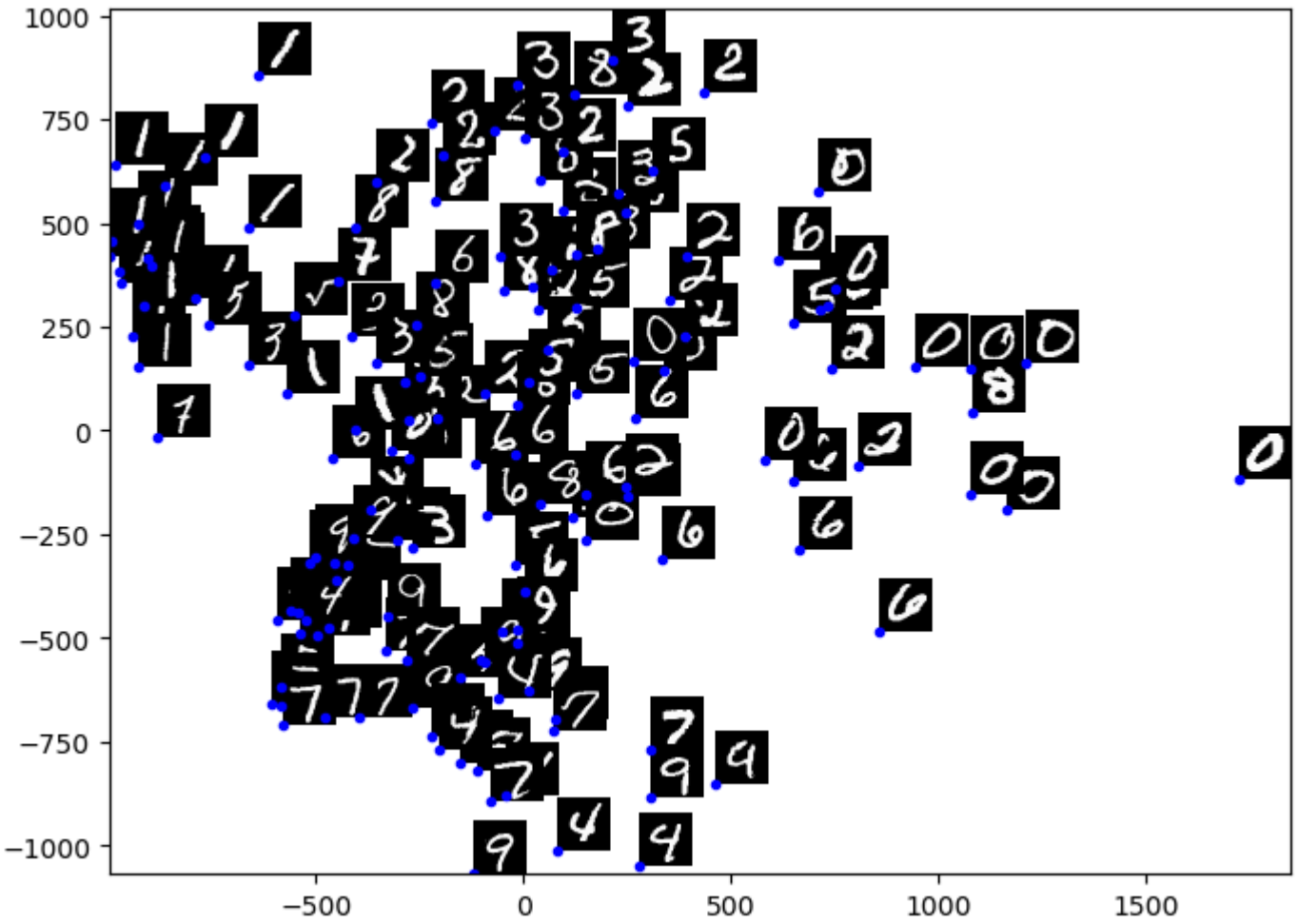
PCA representation of 0..3:

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

In [108…
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(X)
Z = pca.transform(X)
```

In [109…
```
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.

In [113…

```
# 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)
```

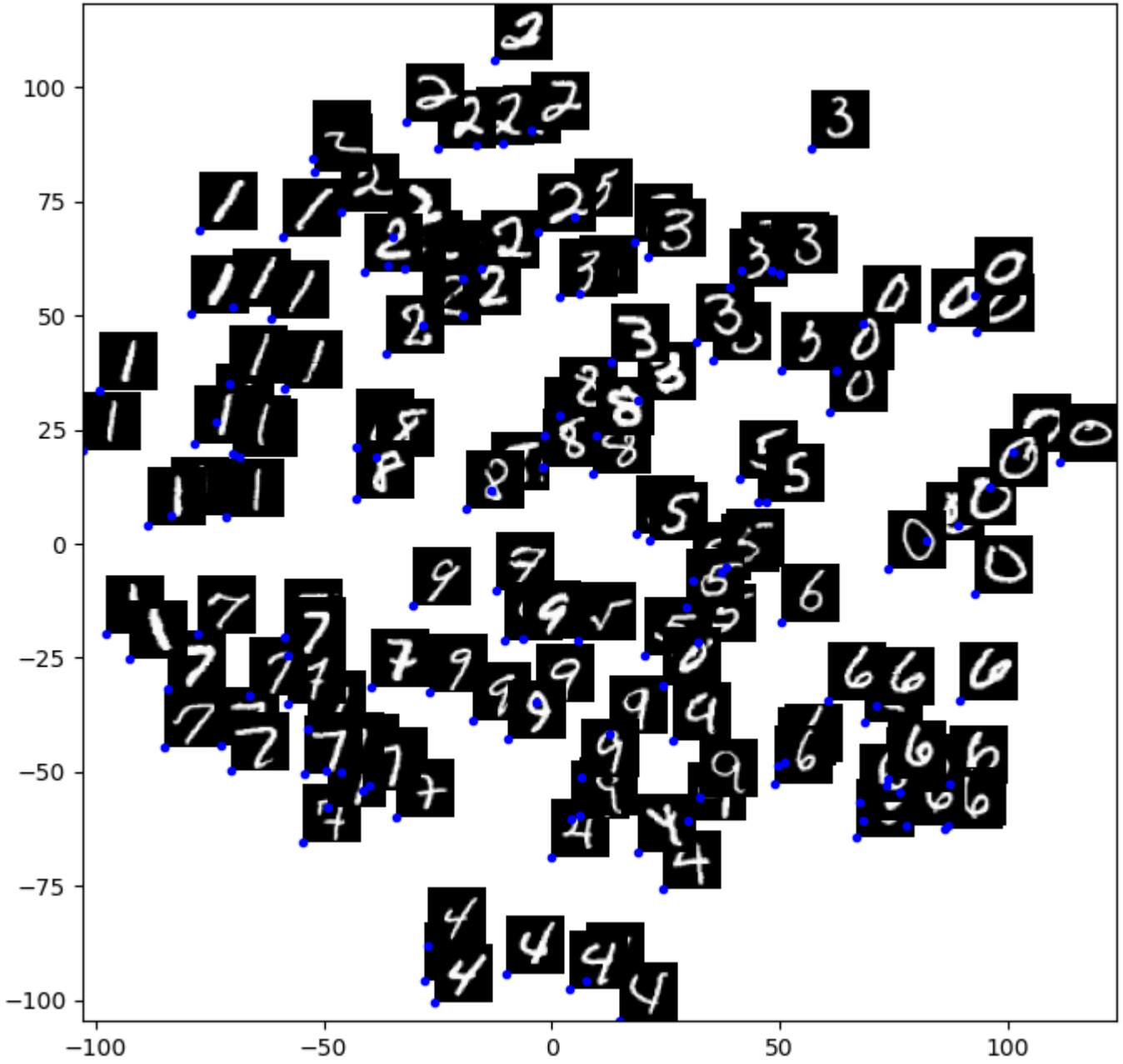
```
[t-SNE] Computing 31 nearest neighbors...
[t-SNE] Indexed 70000 samples in 0.074s...
[t-SNE] Computed neighbors for 70000 samples in 40.083s...
[t-SNE] Computed conditional probabilities for sample 1000 / 70000
[t-SNE] Computed conditional probabilities for sample 2000 / 70000
[t-SNE] Computed conditional probabilities for sample 3000 / 70000
[t-SNE] Computed conditional probabilities for sample 4000 / 70000
[t-SNE] Computed conditional probabilities for sample 5000 / 70000
[t-SNE] Computed conditional probabilities for sample 6000 / 70000
[t-SNE] Computed conditional probabilities for sample 7000 / 70000
[t-SNE] Computed conditional probabilities for sample 8000 / 70000
[t-SNE] Computed conditional probabilities for sample 9000 / 70000
[t-SNE] Computed conditional probabilities for sample 10000 / 70000
[t-SNE] Computed conditional probabilities for sample 11000 / 70000
[t-SNE] Computed conditional probabilities for sample 12000 / 70000
[t-SNE] Computed conditional probabilities for sample 13000 / 70000
[t-SNE] Computed conditional probabilities for sample 14000 / 70000
[t-SNE] Computed conditional probabilities for sample 15000 / 70000
[t-SNE] Computed conditional probabilities for sample 16000 / 70000
[t-SNE] Computed conditional probabilities for sample 17000 / 70000
[t-SNE] Computed conditional probabilities for sample 18000 / 70000
[t-SNE] Computed conditional probabilities for sample 19000 / 70000
[t-SNE] Computed conditional probabilities for sample 20000 / 70000
[t-SNE] Computed conditional probabilities for sample 21000 / 70000
[t-SNE] Computed conditional probabilities for sample 22000 / 70000
[t-SNE] Computed conditional probabilities for sample 23000 / 70000
[t-SNE] Computed conditional probabilities for sample 24000 / 70000
[t-SNE] Computed conditional probabilities for sample 25000 / 70000
[t-SNE] Computed conditional probabilities for sample 26000 / 70000
[t-SNE] Computed conditional probabilities for sample 27000 / 70000
[t-SNE] Computed conditional probabilities for sample 28000 / 70000
[t-SNE] Computed conditional probabilities for sample 29000 / 70000
[t-SNE] Computed conditional probabilities for sample 30000 / 70000
[t-SNE] Computed conditional probabilities for sample 31000 / 70000
[t-SNE] Computed conditional probabilities for sample 32000 / 70000
[t-SNE] Computed conditional probabilities for sample 33000 / 70000
[t-SNE] Computed conditional probabilities for sample 34000 / 70000
[t-SNE] Computed conditional probabilities for sample 35000 / 70000
[t-SNE] Computed conditional probabilities for sample 36000 / 70000
[t-SNE] Computed conditional probabilities for sample 37000 / 70000
[t-SNE] Computed conditional probabilities for sample 38000 / 70000
```

```
[t-SNE] Computed conditional probabilities for sample 39000 / 70000
[t-SNE] Computed conditional probabilities for sample 40000 / 70000
[t-SNE] Computed conditional probabilities for sample 41000 / 70000
[t-SNE] Computed conditional probabilities for sample 42000 / 70000
[t-SNE] Computed conditional probabilities for sample 43000 / 70000
[t-SNE] Computed conditional probabilities for sample 44000 / 70000
[t-SNE] Computed conditional probabilities for sample 45000 / 70000
[t-SNE] Computed conditional probabilities for sample 46000 / 70000
[t-SNE] Computed conditional probabilities for sample 47000 / 70000
[t-SNE] Computed conditional probabilities for sample 48000 / 70000
[t-SNE] Computed conditional probabilities for sample 49000 / 70000
[t-SNE] Computed conditional probabilities for sample 50000 / 70000
[t-SNE] Computed conditional probabilities for sample 51000 / 70000
[t-SNE] Computed conditional probabilities for sample 52000 / 70000
[t-SNE] Computed conditional probabilities for sample 53000 / 70000
[t-SNE] Computed conditional probabilities for sample 54000 / 70000
[t-SNE] Computed conditional probabilities for sample 55000 / 70000
[t-SNE] Computed conditional probabilities for sample 56000 / 70000
[t-SNE] Computed conditional probabilities for sample 57000 / 70000
[t-SNE] Computed conditional probabilities for sample 58000 / 70000
[t-SNE] Computed conditional probabilities for sample 59000 / 70000
[t-SNE] Computed conditional probabilities for sample 60000 / 70000
[t-SNE] Computed conditional probabilities for sample 61000 / 70000
[t-SNE] Computed conditional probabilities for sample 62000 / 70000
[t-SNE] Computed conditional probabilities for sample 63000 / 70000
[t-SNE] Computed conditional probabilities for sample 64000 / 70000
[t-SNE] Computed conditional probabilities for sample 65000 / 70000
[t-SNE] Computed conditional probabilities for sample 66000 / 70000
[t-SNE] Computed conditional probabilities for sample 67000 / 70000
[t-SNE] Computed conditional probabilities for sample 68000 / 70000
[t-SNE] Computed conditional probabilities for sample 69000 / 70000
[t-SNE] Computed conditional probabilities for sample 70000 / 70000
[t-SNE] Mean sigma: 333.847005
[t-SNE] KL divergence after 250 iterations with early exaggeration: 107.839424
[t-SNE] KL divergence after 1000 iterations: 3.133785
```

In [115…

```
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()
```


100
75
50
25
0
−25
−50
−75
−100
−100
−50
0
50
100

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.)

DISCUSS: Both methods can form relatively clear clusters, where each number is clustered in its own region and the positions of the regions divided by the numbers are also similar. The difference is that TSNE is distributed more evenly in low dimensional space, such as when different numbers are distributed more evenly. The degree of overlap between different classes of numbers in PCA will be higher. The reason is that PCA is a linear dimensionality reduction method that can only find linear principal components in high-dimensional space, and cannot capture nonlinear relationships well, resulting in different categories appearing in the same region. TSNE emphasizes the local similarity between points, attempting to maximize the similarity between adjacent points in high-dimensional and low dimensional spaces, thus forming a uniform cluster distribution.

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

DISCUSS: TSNE only has a fit_transform function because it is a non-linear, non-parametric algorithm designed to preserve local relationships in high-dimensional data. It maps data points to a lower-dimensional space through iterative optimization, without learning a global transformation. As a result, TSNE cannot apply the same transformation to new data without re-running the entire algorithm. In contrast, PCA is a linear, parametric method that computes principal components based on variance. Once fitted, PCA learns a global transformation matrix that can be consistently applied to both the original and new data, allowing it to separate the fitting (fit) and transformation (transform) steps.

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.

This assignment is completed independently by me.