MNIST_Image_Reconstruction

July 30, 2024

1 MNIST Image Reconstruction

[ ]:
```
from sklearn.decomposition import PCA
import numpy as np
from matplotlib import pyplot as plt
from ipywidgets import interact
from keras.datasets import mnist
```

1.1 Load the digit image data MNIST

Each image is a digit with image size 28×28

[ ]:
```
(train_X, train_y) = mnist.load_data()[0]
```

Because the original data set is very large, we only use a subset of the data set here.

[ ]:
```
digits_data = train_X[:5000].reshape(5000, -1)
digits_target = train_y[:5000]
digits_data.shape, digits_target.shape
```

[ ]:
```
((5000, 784), (5000,))
```

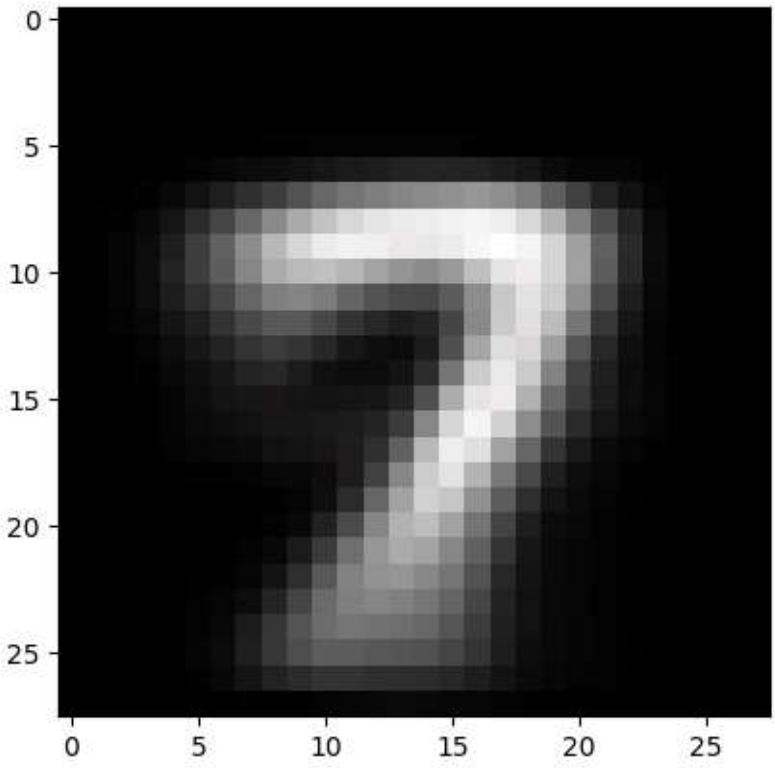
1.2 Part I: Visualize the Principal Components

[ ]:
```
# Favorite number
num = 7
X = digits_data[digits_target == num]
print(X.shape)
```

```
(550, 784)
```

Let's show the mean image of your selected digit $\bar{X} = \sum_{i=1}^{n} X^{(i)}$

[ ]:
```
X_bar = X.mean(axis=0)
plt.imshow(X_bar.reshape((28, 28)), cmap='gray')
plt.show()
```



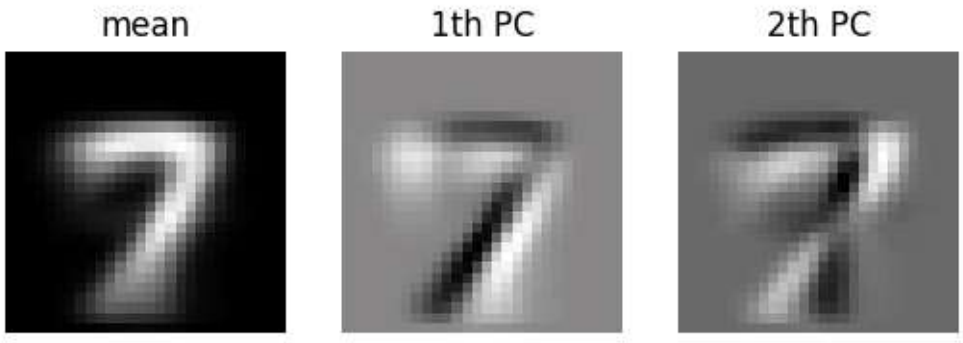
Visualize w_1 and w_2. They are both in $\mathbb{R}^{784}$ and can be visualized as images.

They are the eigenvectors corresponding to the largest and second largest eigenvalues of the covariance matrix. Equivalently, they are right singular vectors corresponding to the largest and second largest singular value of the **centered** data.

Therefore, you can use SVD or eigenvalue decomposition, or use the `PCA` method in `scikit-learn`.

[ ]:
```python
def show_principal_components(k = 2):

    global X, X_bar
    X_centered = X - X_bar
    W = PCA(n_components=k).fit(X_centered).components_.T

    fig, ax = plt.subplots(1, k + 1)
    ax[0].imshow(X_bar.reshape((28, 28)), cmap="gray")
    ax[0].set_title("mean")
    ax[0].axis("off")
    for i in range(k):
        ax[i+1].imshow(W[:, i].reshape((28, 28)), cmap="gray")
        ax[i+1].set_title(f"{i+1}th PC")
        ax[i+1].axis("off")
    plt.show(fig)
```

[ ]:
```
show_principal_components(k=2)
```



[ ]:
```
interact(show_principal_components)
```

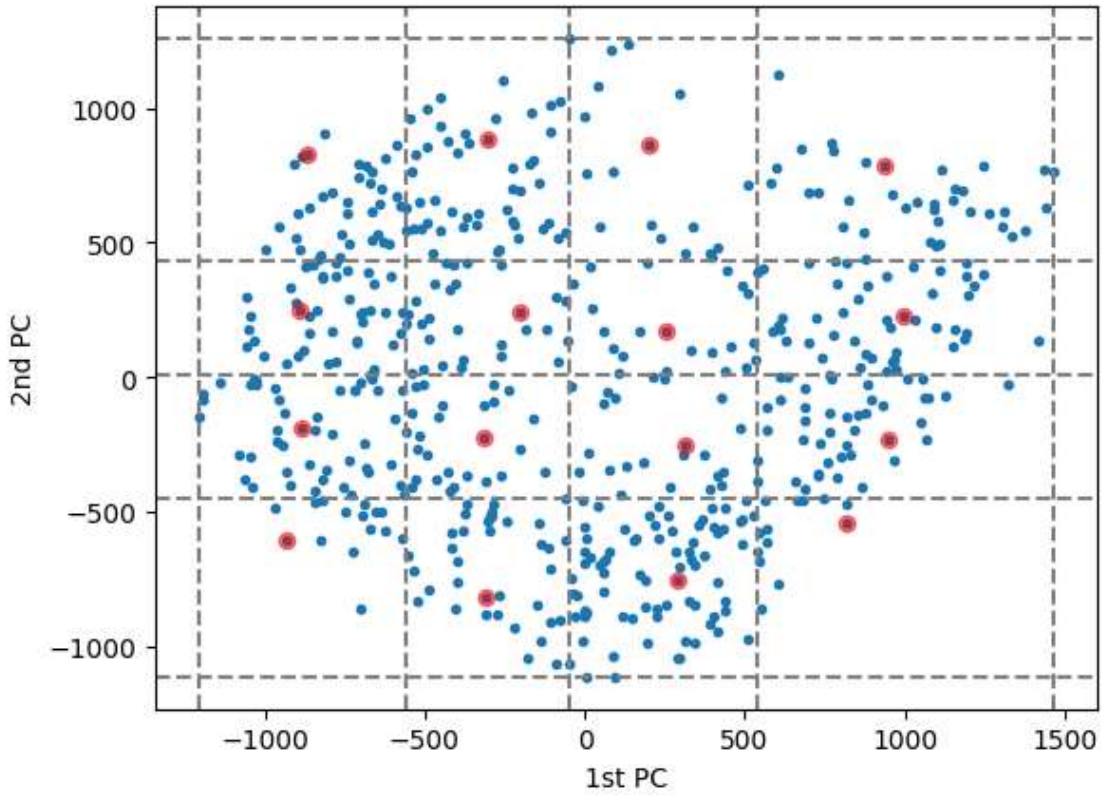
```
interactive(children=(IntSlider(value=2, description='k', max=6, min=-2),␣
↪Output()), _dom_classes=('widget-int…
```

[ ]:
```
<function __main__.show_principal_components(k=2)>
```

Visualize the projection scores

[ ]:
```
from scipy.spatial.distance import cdist

pca = PCA(n_components=2)
Z = pca.fit_transform(X)
plt.plot(Z[:,0], Z[:,1], '.')
X_grid = np.quantile(Z[:,0], [0, 0.25, 0.5, 0.75, 1])
Y_grid = np.quantile(Z[:,1], [0, 0.25, 0.5, 0.75, 1])
xv, yv = np.meshgrid(X_grid, Y_grid)

marked_points = []

for i in range(len(X_grid) - 1):
    for j in range(len(Y_grid) - 1):
        midpoint_x = (X_grid[i] + X_grid[i+1]) / 2
        midpoint_y = (Y_grid[j] + Y_grid[j+1]) / 2
        midpoint = np.array([midpoint_x, midpoint_y]).reshape(1, -1)

        distances = cdist(midpoint, Z).flatten()

        closest_point_idx = np.argmin(distances)
        closest_point = Z[closest_point_idx]

        marked_points.append(closest_point)
        plt.plot(closest_point[0], closest_point[1], 'ro', alpha = 0.5)
```

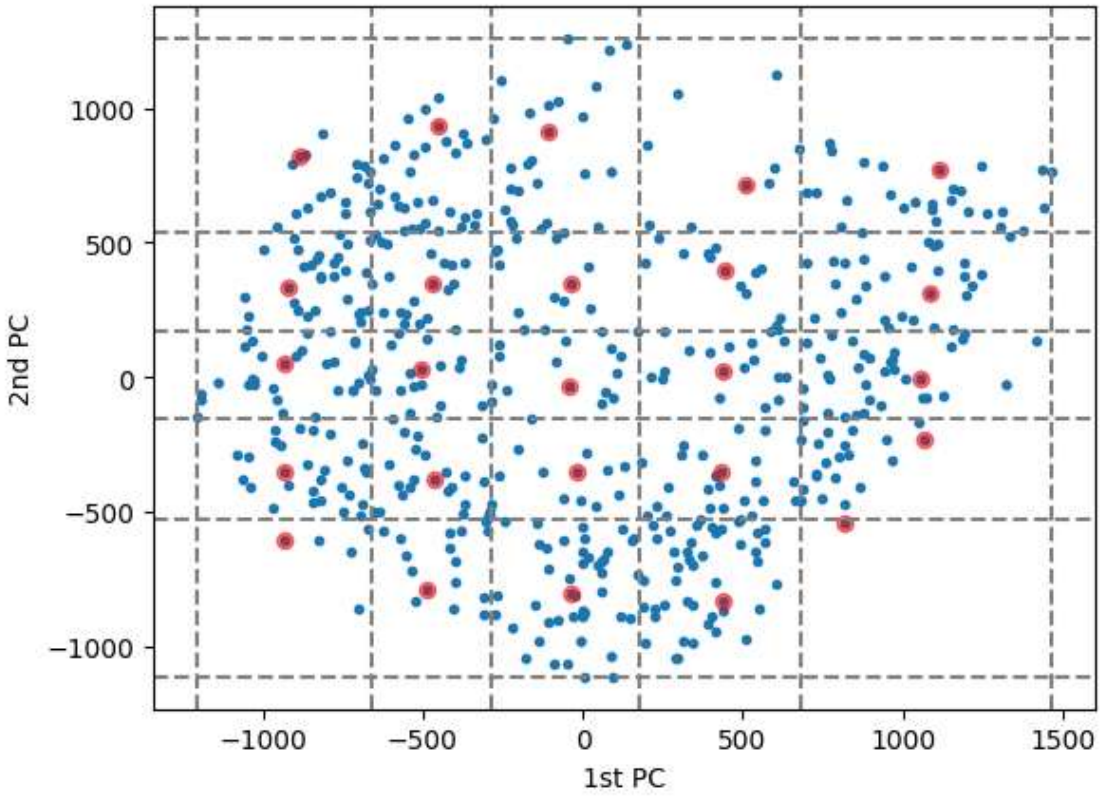
```
for x in X_grid:
    plt.axvline(x, color='gray', linestyle='--')
for y in Y_grid:
    plt.axhline(y, color='gray', linestyle='--')

plt.xlabel("1st PC")
plt.ylabel("2nd PC")
plt.show()
```



Ended up with a 4x4 plot, not a 5x5 plot. So the following plot is the same, but using np.linspace rather than np.quantile to make a 5x5 plot. I am using both plots for my answer here, since one or the other may be the correct interpretation of what is wanted for this problem.

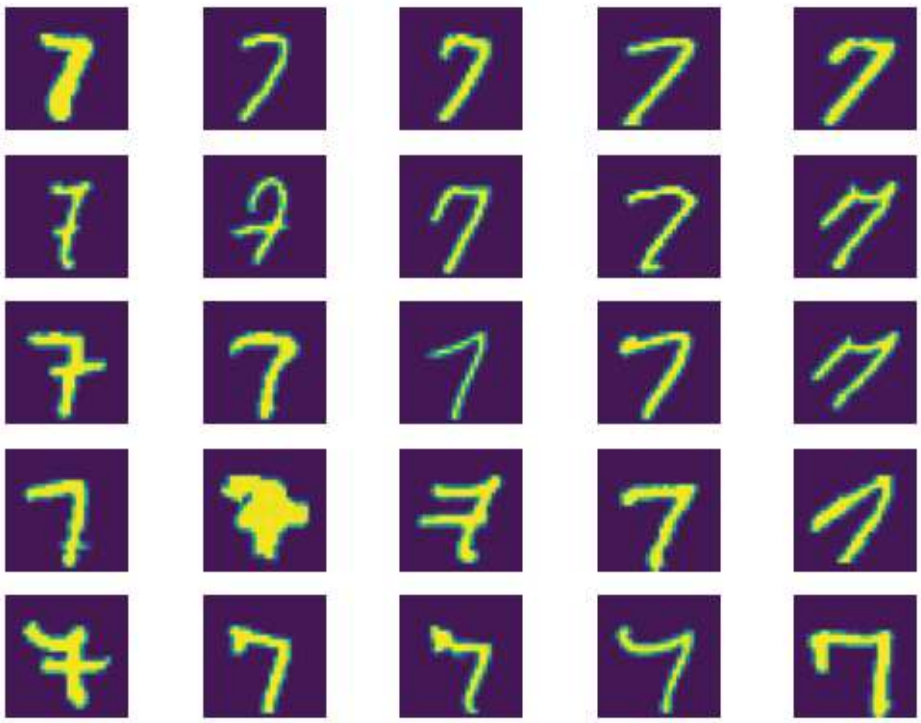
[ ]:
```
pca = PCA(n_components=2)
Z = pca.fit_transform(X)

plt.plot(Z[:,0], Z[:,1], '.')
X_grid = np.quantile(Z[:,0], np.linspace(0, 1, 6))
Y_grid = np.quantile(Z[:,1], np.linspace(0, 1, 6))
```

```
xv, yv = np.meshgrid(X_grid, Y_grid)

marked_points2 = []

for i in range(len(X_grid) - 1):
    for j in range(len(Y_grid) - 1):
        midpoint_x = (X_grid[i] + X_grid[i+1]) / 2
        midpoint_y = (Y_grid[j] + Y_grid[j+1]) / 2
        midpoint = np.array([midpoint_x, midpoint_y]).reshape(1, -1)

        distances = cdist(midpoint, Z).flatten()

        closest_point_idx = np.argmin(distances)
        closest_point = Z[closest_point_idx]

        marked_points2.append(closest_point)
        plt.plot(closest_point[0], closest_point[1], 'ro', alpha = 0.5)

for x in X_grid:
    plt.axvline(x, color='gray', linestyle='--')
for y in Y_grid:
    plt.axhline(y, color='gray', linestyle='--')
plt.xlabel("1st PC")
plt.ylabel("2nd PC")
plt.show()
```



Making a figure with 5*5 subplots, each one displays the original image corresponds to the marked data point in the previous plot.

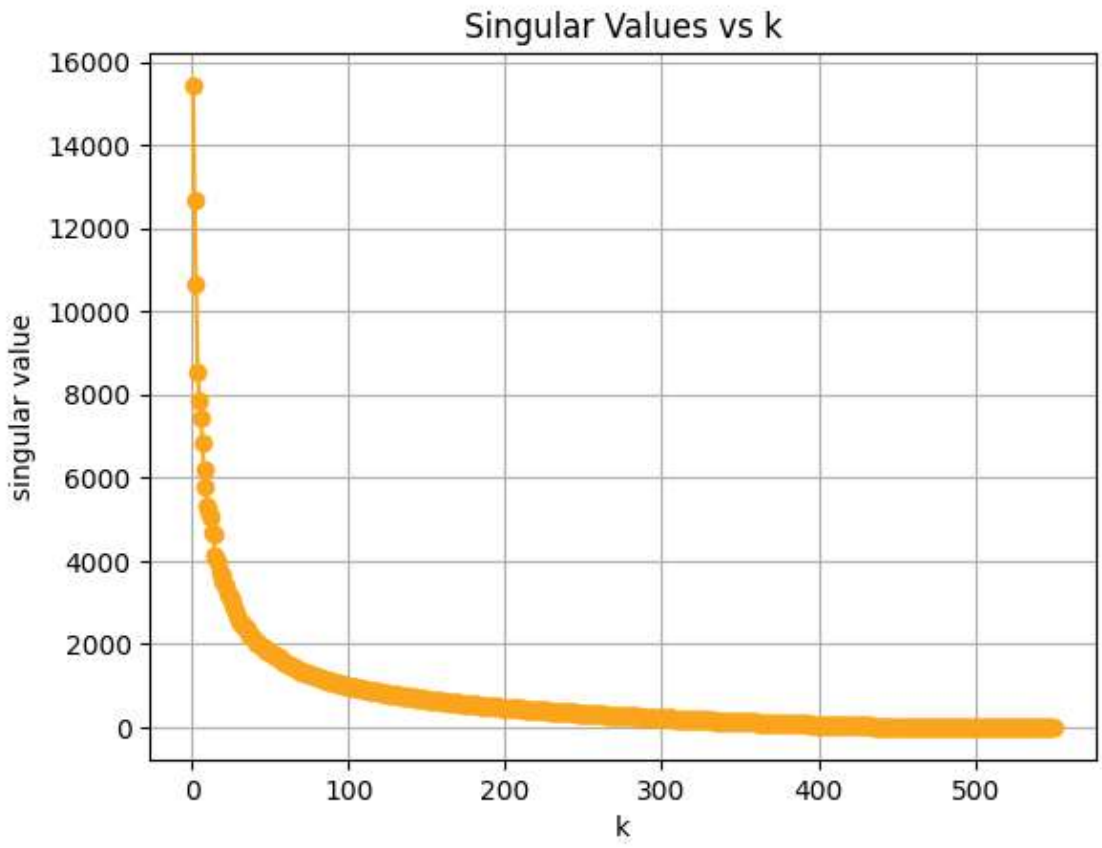
[ ]:
```
fig, ax = plt.subplots(5, 5)
def plot_images(images, marked_points, ax):
    for i, ax_row in enumerate(ax):
        for j, ax_col in enumerate(ax_row):
            idx = i * 5 + j
            if idx < len(marked_points):
                closest_point_idx = np.argmin(cdist([marked_points[idx]], Z))
                ax_col.imshow(images[closest_point_idx].reshape(28,28))
                ax_col.axis('off')

plot_images(X, marked_points2, ax)
plt.show(fig)
```



Each image corresponds to the point that is closest to the middle of each cell. These points are marked in red. The middle is the most average of the data points. Each point represents an image of the number 7. This means that each of the 25 images we see are typical for the number. Furthermore, each image is from a cell of the scatterplot, so they represent that cell's average image. The images from the last two columns do not have a line through the number 7 and are relatively thin. The first three columns have more images with the number 7 with a line through it and are thicker. This is especially true for the first image in the first row and the fourth image in the second row.

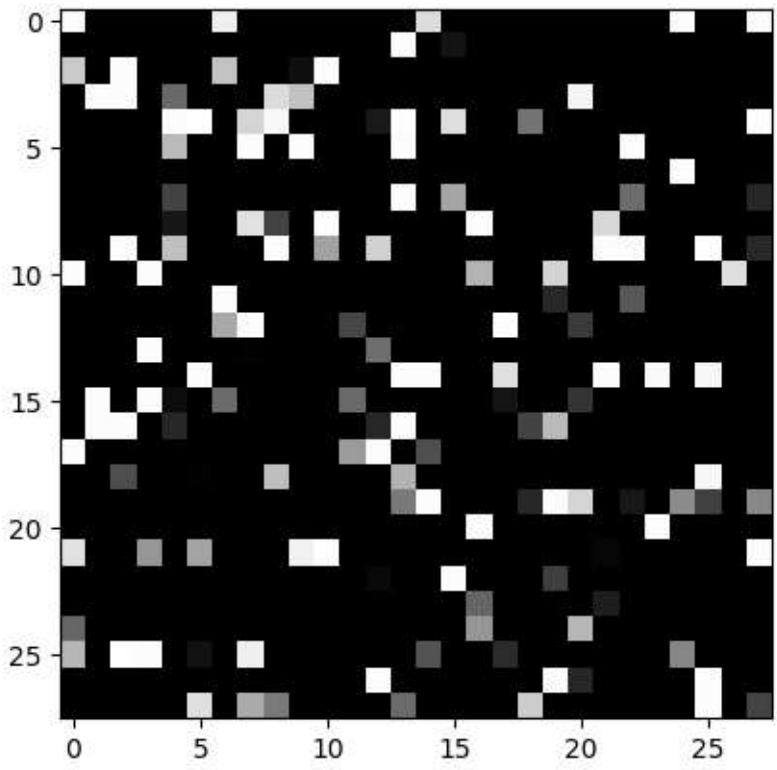
singular value of X changing with k

[ ]:
```
pca = PCA()
pca.fit(X - X_bar)
singular_values = pca.singular_values_
k = np.arange(1, len(singular_values) + 1)
plt.plot(k, singular_values, marker='o', color='orange')
plt.xlabel('k')
plt.ylabel('singular value')
plt.title('Singular Values vs k')
plt.grid(True)
plt.show()
```



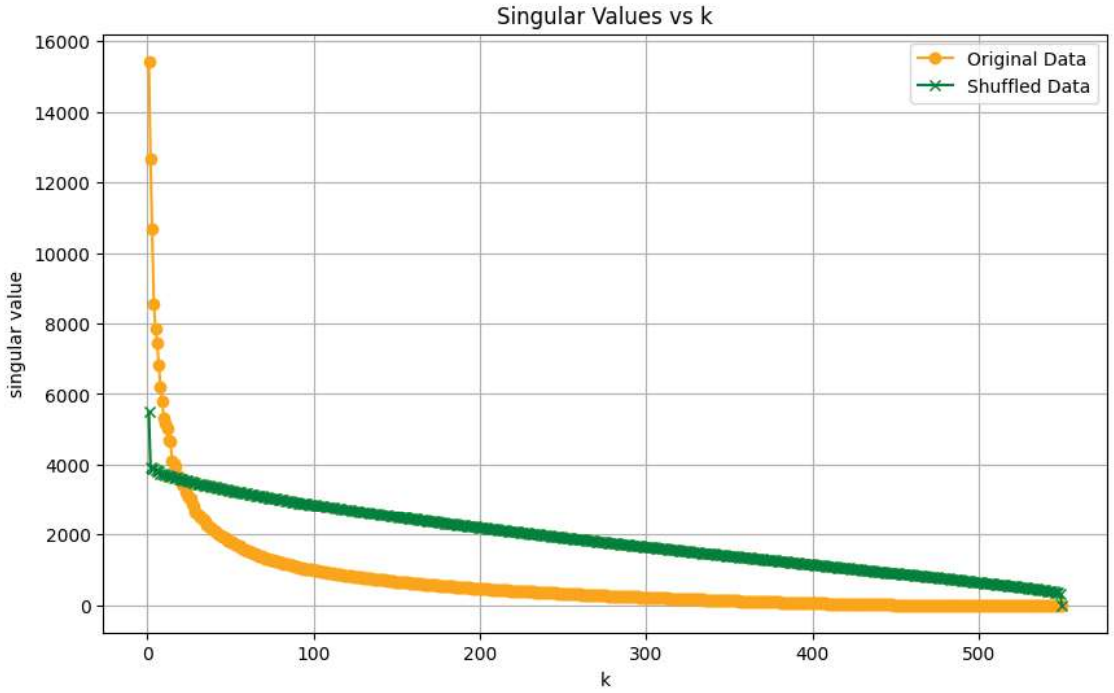
For comparison, let's now add another curve. For each image in X, Randomly shuffle the entries. The image should look like random noise after the shuffling.

[ ]:
```
X_shuffle = X.copy()
for i in range(X_shuffle.shape[0]):
    np.random.shuffle(X_shuffle[i])
plt.imshow(X_shuffle[0].reshape((28, 28)), cmap="gray")
plt.show()
```



Then add another "singular value vs k" curve for `X_shuffle`, with different color. Add a label for each curve.

[ ]:
```
pca_shuffled = PCA()
pca_shuffled.fit(X_shuffle - X_bar)
singular_values_shuffled = pca_shuffled.singular_values_
k = np.arange(1, len(singular_values) + 1)
plt.figure(figsize=(10, 6))
plt.plot(k, singular_values, marker='o', label='Original Data', color='orange')
plt.plot(k, singular_values_shuffled, marker='x', label='Shuffled Data',␣
↪color='green')
plt.xlabel('k')
plt.ylabel('singular value')
plt.title('Singular Values vs k')
plt.legend()
plt.grid(True)
plt.show()
```



Kernel Density Estimation – tuning the `bandwidth`

We now apply PCA to the entire data set with $ k = 50$. `Z` is now the projection scores with shape (n, 15)

- Compute the kernel density estimator using `sklearn.neighbors.KernelDensity`. Here, let's simply use the Gaussian kernel and the Euclidean distance, and explore the effect of the `bandwidth` parameter. Discover the difference between using two different bandwidths, 0.1 and 200.
- Draw 16 new sample points from each fitted density using `KernelDensity.sample` (for comparison purpose, please use the same value for `random_state` for both bandwidths). Of course, these samples are in the space of projection scores. Use `pca.inverse_transform` to reconstruct digit images from these samples. Compare the sample digits drawn from density fitted with large bandwidth and small bandwidth. What are the differences? Could you give some intuitive explanations why you are seeing overlapping digits when `bandwidth` is large?

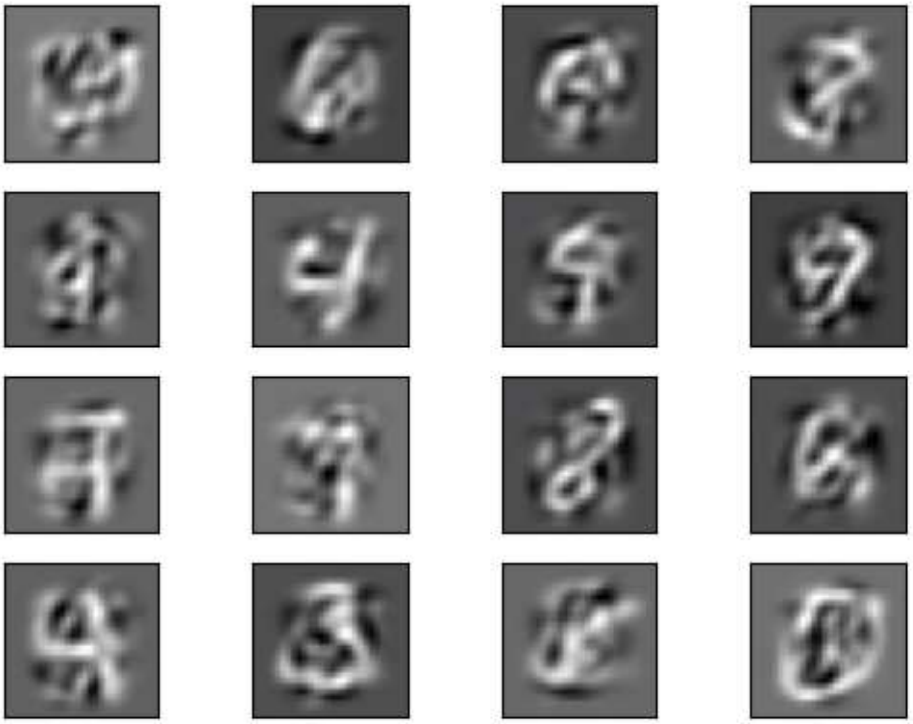
After seeing the differences between the images using a bandwidth of 0.1 and 200, those from the small bandwidth are very clear and easy to read while the large bandwidth is blurry. However, even with the blurriness, I can tell that many images for both bandwidths are the same digit. I think that this happened because the bandwidth is the width of the kernel. So when the only difference between the two is the bandwidth, this means that there are overlapping images because the digits themselves will be around the same. The wide bandwidth captures the overall trend of the data while the small bandwidth shows the clear details, which results in these overlapping images. The large bandwidth captures the same digits as the small bandwidth, along with images that capture the overall trend of the data. By looking closely between the images from the two bandwidths, pretty much every digit is the same, but the one from the large bandwidth shows the digit's general

shape, with some blurriness.

[ ]:
```
from sklearn.neighbors import KernelDensity

k = 50
pca = PCA(n_components=k)
Z = pca.fit_transform(digits_data)

def sample_digits(bandwidth: float = 1.0, n_samples:int = 16) -> np.ndarray:

    kde = KernelDensity(kernel='gaussian', metric = 'euclidean',␣
 ↪bandwidth=bandwidth)
    kde.fit(Z)
    new_Z = kde.sample(n_samples, random_state=12)
    new_digits = pca.inverse_transform(new_Z)
    return new_digits

new_digits_1 = sample_digits(bandwidth=0.1, n_samples=16).reshape((4, 4, -1))
new_digits_2 = sample_digits(bandwidth=200, n_samples=16).reshape((4, 4, -1))
```

[ ]:
```
def show_digits(new_digits: np.ndarray):
    '''
    new_digits have shape (4, 4, 784)
    '''
    fig, ax = plt.subplots(4, 4, subplot_kw=dict(xticks=[], yticks=[]))

    for i in range(4):
        for j in range(4):
            ax[i, j].imshow(new_digits[i, j].reshape(28, 28), cmap='gray')

    plt.show(fig)
```

[ ]:
```
show_digits(new_digits_1)
show_digits(new_digits_2)
```



Use `sklearn.model_selection.GridSearchCV` to select the best bandwidth. Refit the model with the selected bandwidth, then draw another sample of 16 digits using that density. You can search for bandwidth within the choices `np.logspace(-1, 3, 20)`.

`GridSearchCV` (by default) uses the estimator's `score` method to evaluation performance on the test folds. In the case of `KernelDensity`, `score()` gives the log-likelihood of the test data in the estimated density. (Here it is the likelihood under the Gaussian mixture model (GMM))

Note that your CV-selected bandwidth, which maximizes the likelihood on the test folds, does not necessarily look better than the previous choices. Could you (intuitively) explain why?

[ ]:
```
from sklearn.model_selection import GridSearchCV

param_grid = {'bandwidth': np.logspace(-1, 3, 20)}
grid_search = GridSearchCV(KernelDensity(kernel='gaussian', metric='euclidean'),
                           param_grid, cv=5)
grid_search.fit(Z)
cv_selected_bandwidth = grid_search.best_params_['bandwidth']
new_digits_cv = sample_digits(bandwidth=cv_selected_bandwidth, n_samples=16).
 ↪reshape((4, 4, -1))
show_digits(new_digits_cv)
```

The reason why these images do not look better than those generated previously using a bandwidth of 0.1 is because it uses log-likelihood, which aims to explain how well the density function fits the training data. Just because a model has a higher log-likelihood does not mean that the visual

image itself is better. This focuses on statistical fit rather than the visual reality.

1.3 Part II Select the number of clusters using the *Gap statistic* (Tibshirani et al., 2001b).

This paper provides a statistical procedure to formalize the heuristic of the "elbow method", by looking at the gap between the change in within-cluster dispersion and its "expected" version.

The algorithm is described in detail in section 4 in the paper.

$$\widehat{k} = \arg\min_k \{k | \text{Gap}(k) \geq \text{Gap}(k+1) - s_{k+1}\}$$

1.3.1 (a) Implement the algorithm.

- There are two choices of reference distribution mentioned in Section 4. You need to implement both of them. For example, in your function you can specify a parameter called `need_PCA`, if `need_PCA==True`, draw B sample from the reference distribution (a). Otherwise, draw B samples from the reference distribution (b). (By default, set $B = 50$)
- For efficiency, you can specify an upper limit for k. Namely, only calculate Gap(k) for $ 1 k $ `kmax`. If none of the considered k satisfy $ Gap(k) Gap(k + 1) - s_{k+1}$, then output a result that represents $k >$ `kmax` - 1, such as `np.inf` or `None`.
- Besides the selected k, you should also be able to output plots like in Fig. 2 of the paper. Our data `X` in general has $p \geq 2$. Instead of the scatter plot (a) in Fig. 2, make a plot of $\text{Gap}(k) - \text{Gap}(k+1) + s_{k+1}$ against k, so that $\widehat{k}$ will be the smallest k that gives a positive value of that curve. (You can add an horizontal dashed line at $y = 0$)

Some suggestions for making the plots: - Introduce a parameter named `need_plot` to control whether to make the plots, or simply output the selected $\widehat{k}$. - Use `plt.subplots((2,2))`. If you want to adjust the overall figure size, you can set e.g. `figsize=(10, 10)` in `subplot()` - To show the error bars like in Fig 2 (d), you can use `plt.errorbar`. - Specify what is your x-axis and what is your y-axis for each subplot. Make a title if necessary.

[ ]:
```
from sklearn.cluster import KMeans
np.random.seed(12)

'''
returns the number of clusters k selected by the Gap statistic. If the selected␣
 ↪k is larger than kmax, return None.
X has shape (n, p). n is the sample size. p is the number of features.
kmax is maximum k whose Gap(k) will be computed
B is the number of copies from the reference distribution
need_PCA means whether to use the reference distribution (1) or (2)
if need_plot = True, make the required plots. You can use plt.subplots().
'''#

def gap_statistic(X, kmax, B, need_PCA, num_pca = None):
```

```
    if num_pca == None:
        num_pca = min(X.shape)
    (n, p) = X.shape
    if need_PCA:
        pca = PCA(n_components = num_pca)
        X = pca.fit_transform(X)
    gaps = np.zeros(kmax)
    s = np.zeros(kmax)
    for k in range(1, kmax + 1):
        km = KMeans(n_clusters=k, n_init=10)
        km.fit(X)
        ref_disps = np.zeros(B)
        for i in range(B):
            if need_PCA:
                reference = pca.inverse_transform(np.random.rand(n, min(n, p)))
            else:
                reference = np.random.rand(n, p)
            km_ref = KMeans(n_clusters=k, n_init=10)
            km_ref.fit(reference)
            ref_disps[i] = km_ref.inertia_
        gap = np.mean(np.log(ref_disps)) - np.log(km.inertia_)
        sk = np.sqrt(1 + 1 / B) * np.std(np.log(ref_disps))
        gaps[k-1] = gap
        s[k-1] = sk
    return gaps, s

def select_k_by_Gap_statistic(X: np.ndarray, kmax: int = 20, B: int = 50,␣
 ↪need_PCA: bool = False, need_plot: bool = True) -> int:
    gaps, s = gap_statistic(X, kmax, B, need_PCA)
    k_selected = np.inf
    for k in range(1, kmax):
        if gaps[k-1] >= gaps[k] - s[k]:
            k_selected = k
            break

    if need_plot:
        fig, ax = plt.subplots(2, 1, figsize=(10, 10))
        ax[0].plot(range(1, kmax + 1), gaps, marker='o')
        ax[0].set_xlabel('Number of clusters k')
        ax[0].set_ylabel('Gap(k)')
        ax[0].set_title('Gap Statistic for different k')
        ax[1].plot(range(1, kmax), gaps[:-1] - gaps[1:] + s[1:], marker='o')
        ax[1].axhline(y=0, color='r', linestyle='--')
        ax[1].set_xlabel('Number of clusters k')
        ax[1].set_ylabel('Gap(k) - Gap(k+1) + sk+1')
        ax[1].set_title('Gap Difference')
        plt.show()
```

```
    return k_selected if k_selected <= kmax else None
```

1.3.2 (b) Apply the algorithm to the data containing only your favourite digit.

Use the original data as input. Then apply the algorithm, using two different reference distributions, respectively.

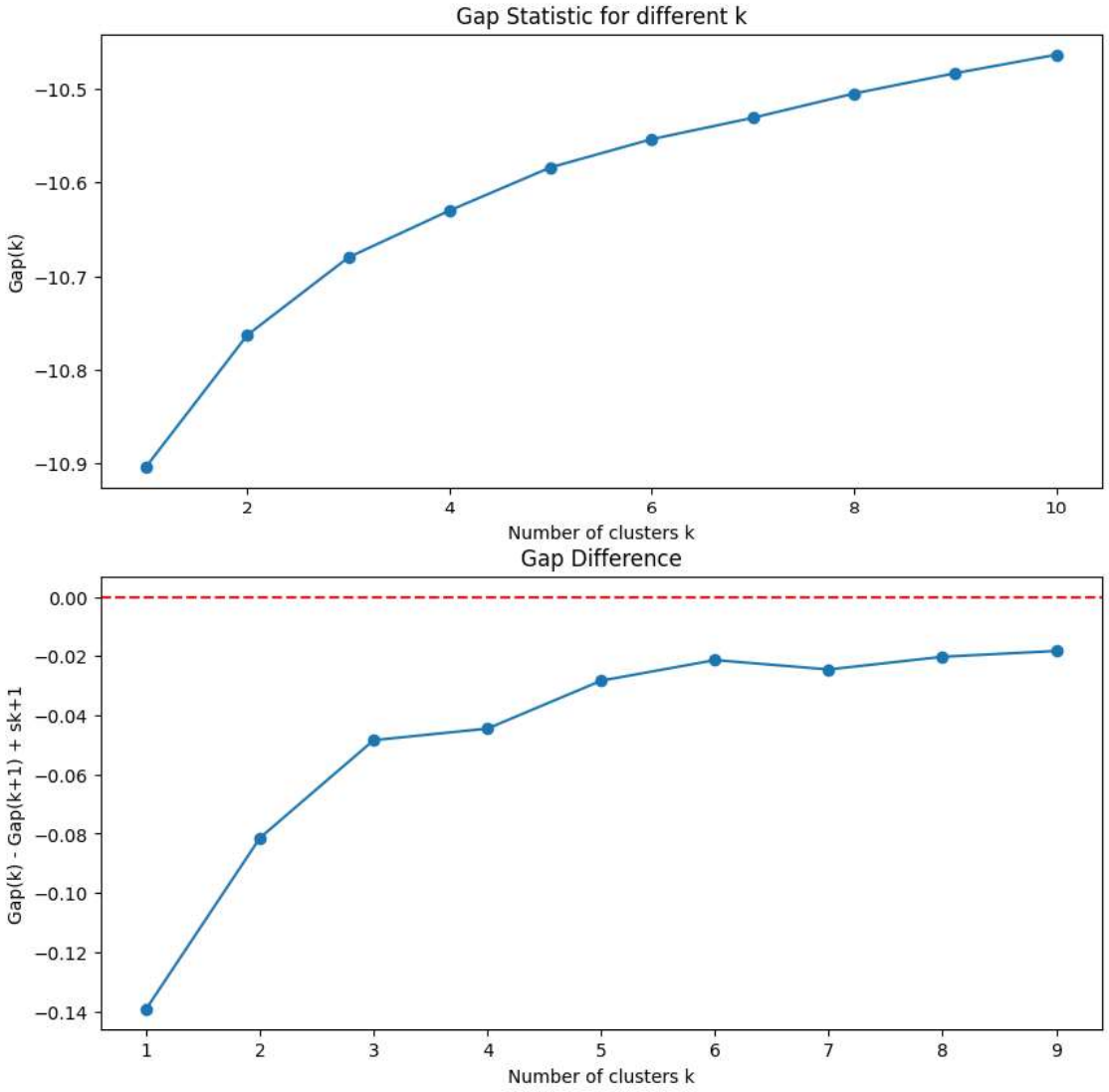
Following is some example code. Feel free to modify it.

[ ]:
```
#Choosing Number 7
num = 7
X = digits_data[digits_target == num]
B = 100
print(X.shape)
```

```
(550, 784)
```

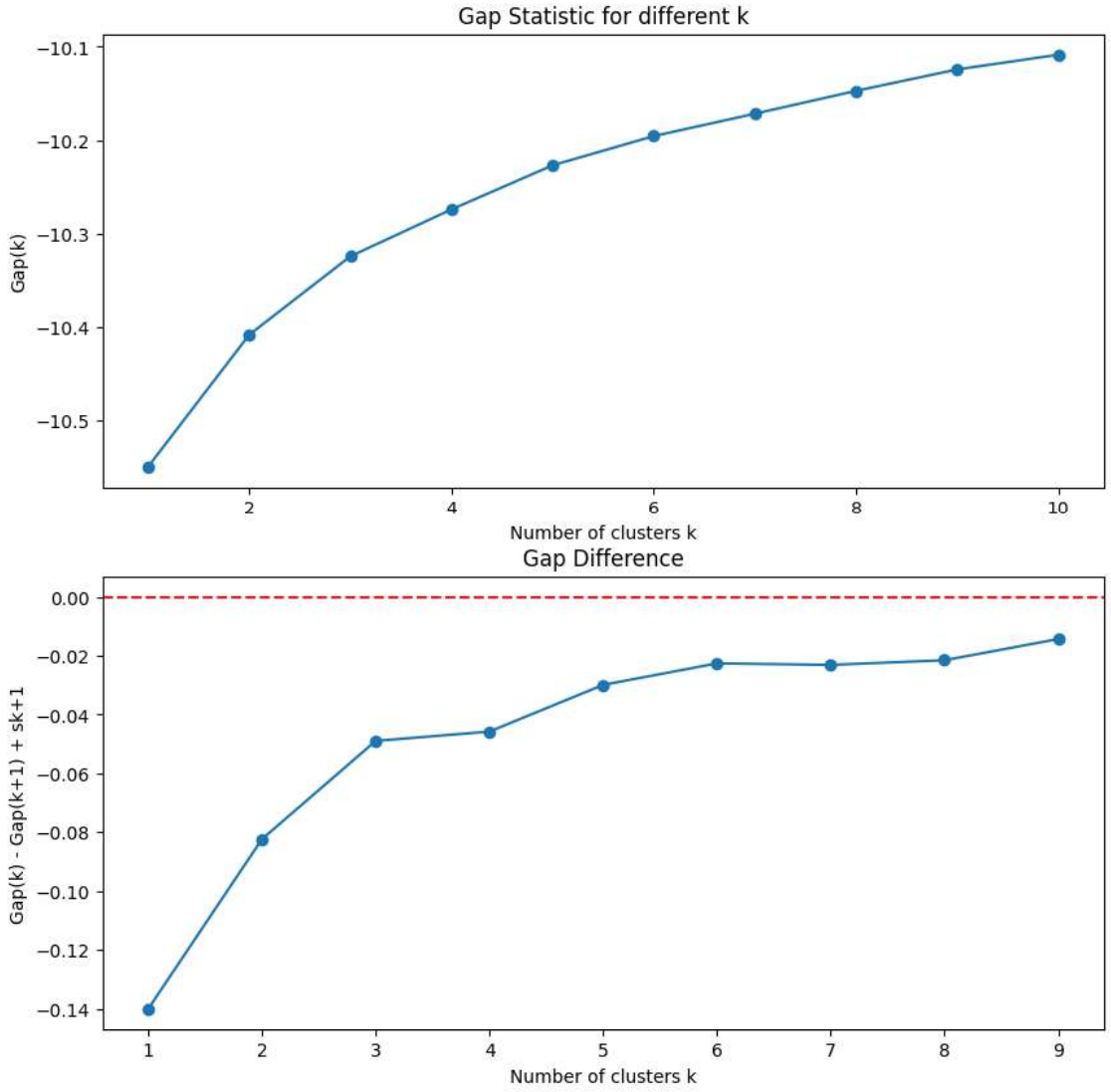
1.3.3 Computing Gap Statistic with reference distribution generated from Singular Value Projections.

[ ]:
```
select_k_by_Gap_statistic(X, kmax=10, B=50, need_PCA=True, need_plot=True)
```



1.3.4 Computing Gap Statistic with reference distribution estimated with Monte Carlo Simulations B = 50.

[ ]:
```
select_k_by_Gap_statistic(X, kmax=10, B=50, need_PCA=False, need_plot=True)
```



My guess of the output $\hat{k}$ is 10, and the optimal number of clusters that maximizes Gap(k) is indeed 10.

The minima and maxima occur at a different number of clusters for each reference distribution. In addition, the maximum gap distance with the uniform distribution is much greater than that with PCA. This is because the PCA method detects more subtle differences and can select a higher number of clusters than the uniform distribution. The PCA distribution does a better job at capturing the structure of complex, correlated data.

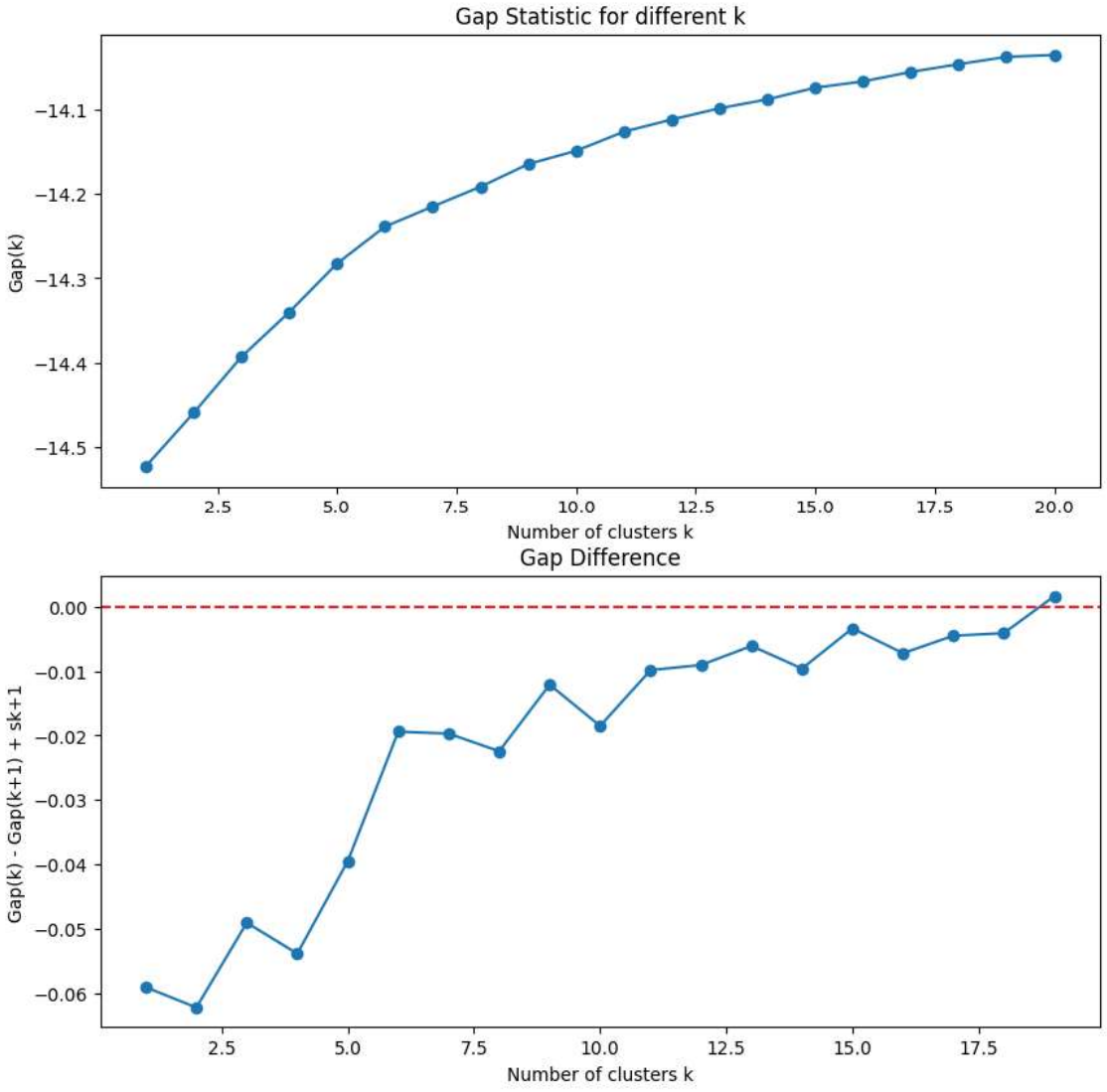
1.3.5 We now apply the algorithm to the entire data set. But this time we will first transform the data by PCA.

For example, we first project the data to 10 principal component scores.

[ ]:
```
pca_complete_dataset = PCA(n_components=10)
pca_complete_dataset.fit(digits_data)
X = pca_complete_dataset.transform(digits_data)
print(X.shape)
```

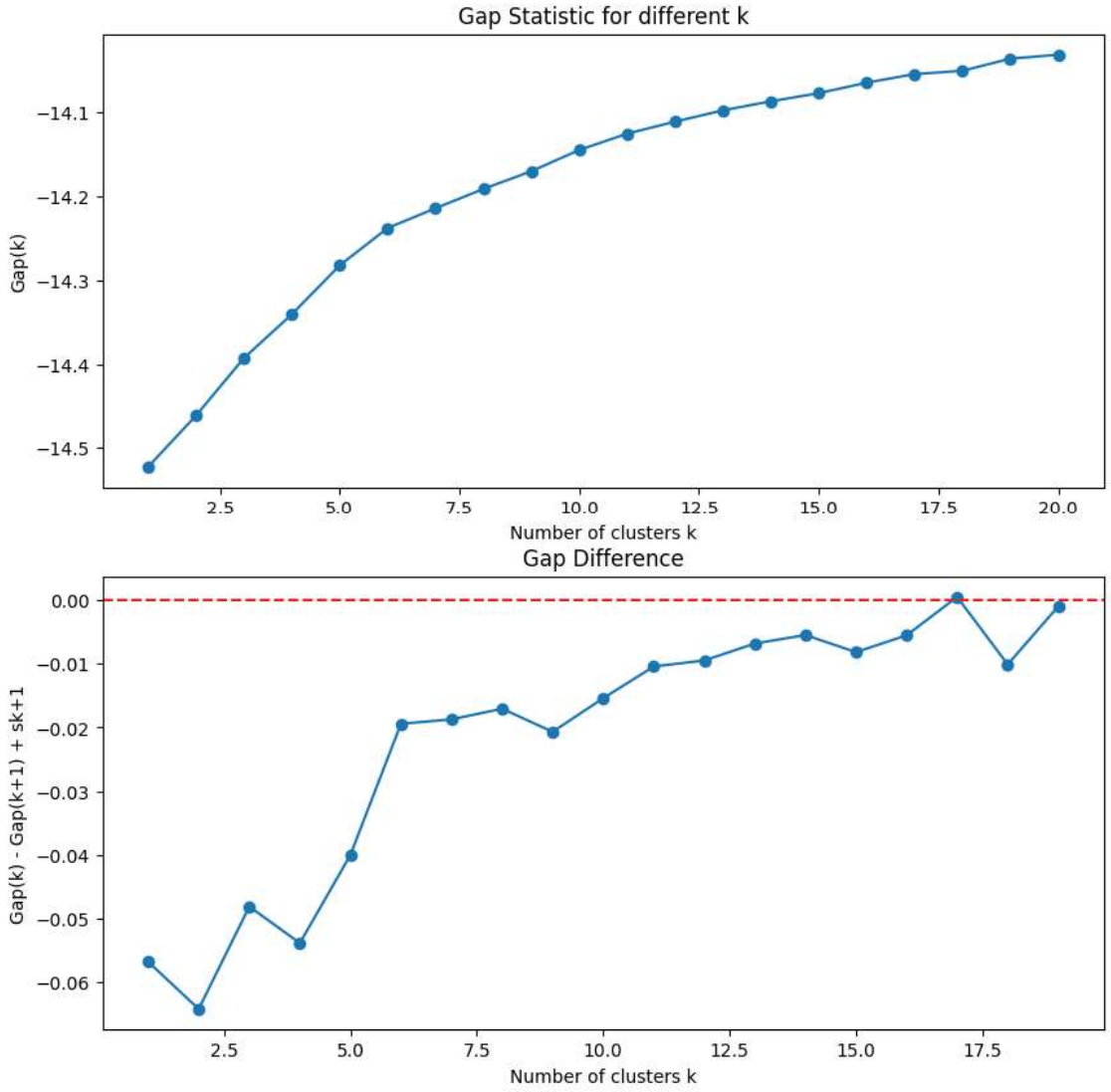
```
(5000, 10)
```

[ ]:
```
select_k_by_Gap_statistic(X, kmax=20, B=B, need_PCA=True, need_plot=True)
```



[ ]:
```
19
```

[ ]:
```
select_k_by_Gap_statistic(X, kmax=20, B=B, need_PCA=False, need_plot=True)
```



[ ]:
```
17
```

Note that now we obtain exactly the same results when using the two different reference distributions. Why?

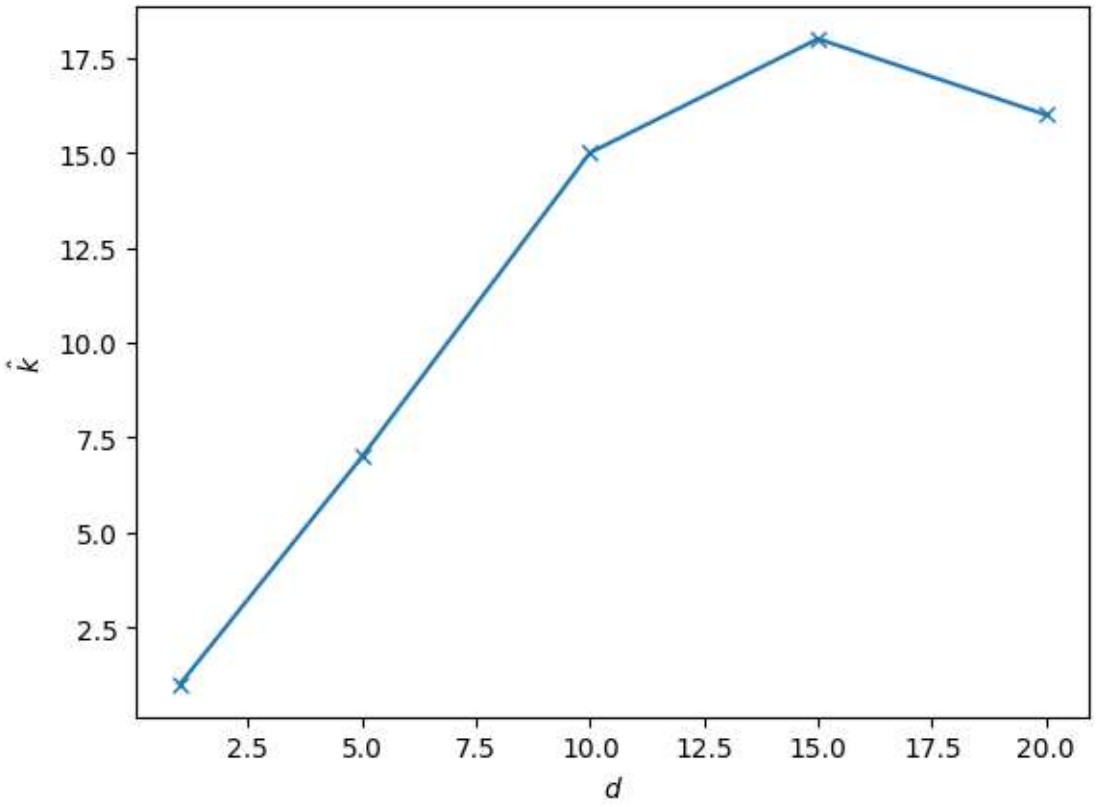
PCA normalizes the variance structure of the data set. Both the uniform distribution and the PCA reference distribution generate data points that are uniform in the feature space. The Gap statistic measures the difference in number of clusters in a set of data, and both reference distributions span the feature space uniformly. Therefore, we would obtain exactly the same results.

Exploring how the PCA projection dimension d affects the $\hat{k}$ selected by the Gap statistic.

[ ]:
```
d_list = [1, 5, 10, 15, 20, 25, 40, 60]
k_hat_list = []
for d in d_list:
```

```
    pca_complete_dataset = PCA(n_components=d)
    pca_complete_dataset.fit(digits_data)
    X = pca_complete_dataset.transform(digits_data[:1000, ])
    k_hat = select_k_by_Gap_statistic(X, kmax=20, B=50, need_PCA=False,␣
 ↪need_plot=False)
    k_hat_list.append(k_hat)
    print(f"PCA with {d} components. Selected k: {k_hat}")

plt.plot(d_list, k_hat_list, 'x-')
plt.xlabel(r'$d$')
plt.ylabel(r'$\hat k$')
plt.show()
```

```
PCA with 1 components. Selected k: 1
PCA with 5 components. Selected k: 7
PCA with 10 components. Selected k: 15
PCA with 15 components. Selected k: 18
PCA with 20 components. Selected k: 16
PCA with 25 components. Selected k: None
PCA with 40 components. Selected k: None
PCA with 60 components. Selected k: None
```



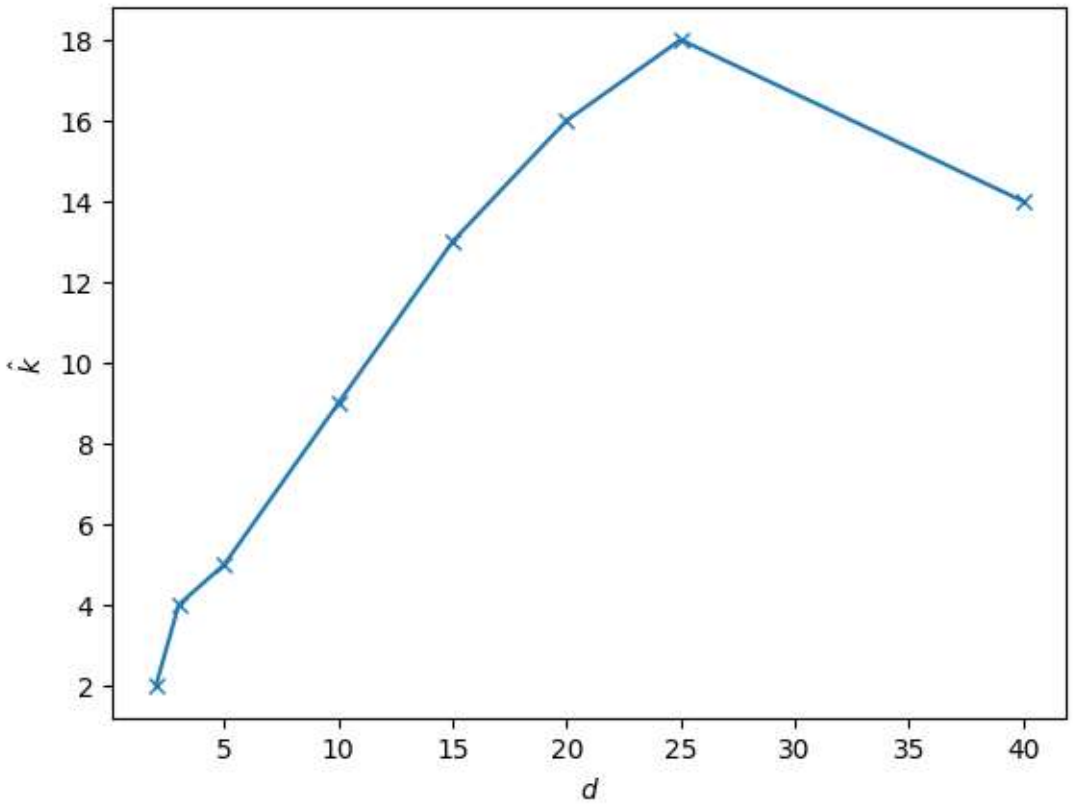
This change in $\hat{k}$ is directly correlated with the dimension of the PCA projections. We understand that PCA reduces the dimensionality of the data set all while trying its best to maintain the variance. Different values of d represent different details in the data. For example: When $d = 2$, much less variance is captured which can affect the principle components ability to represent the data. In contrast, when $d = x$ where x is very close to p (dimension of the data set), we find that the sample clusters are much more closer to the true clusters; higher d implies more dimensions/date, which in turn explains more variance and retains more of the data's original structure

on the projection scores of your favorite digit. Again, make a plot of $\hat{k}$ against d.

Consider d in `[2, 3, 5, 10, 15, 20, 25, 40]`. Note that we use the PCA model *fitted on the entire data set*, but only use the projection scores of your favorite digit.

[ ]:
```
num = 7
d_list = [2, 3, 5, 10, 15, 20, 25, 40]
k_hat_list = []
for d in d_list:
    pca_complete_dataset = PCA(n_components=d)
    pca_complete_dataset.fit(digits_data)
    X = pca_complete_dataset.transform(digits_data[digits_target == num, ]) #␣
 ↪reduce the sample size for efficiency
    k_hat = select_k_by_Gap_statistic(X, kmax=20, B=50, need_PCA=False,␣
 ↪need_plot=False)
    k_hat_list.append(k_hat)
    print(f"PCA with {d} components. Selected k: {k_hat}")

plt.plot(d_list, k_hat_list, 'x-')
plt.xlabel(r'$d$')
plt.ylabel(r'$\hat k$')
plt.show()
```

```
PCA with 2 components. Selected k: 2
PCA with 3 components. Selected k: 4
PCA with 5 components. Selected k: 5
PCA with 10 components. Selected k: 9
PCA with 15 components. Selected k: 13
PCA with 20 components. Selected k: 16
PCA with 25 components. Selected k: 18
PCA with 40 components. Selected k: 14
```



Intuitively explain why you see $\hat{k}$ change with d in this way.

Similar to my response above, the more variance we have in a data set, the more we can explain and differentiate. The reason for more variability in clusters is the reduction of dimensionality decrease towards the true dimension p.

[ ]:
```
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic
!jupyter nbconvert --to pdf /content/Team_18_Final_Project_STA142B_S24.ipynb
```

```
Reading package lists… Done
Building dependency tree… Done
Reading state information… Done
The following additional packages will be installed:
  dvisvgm fonts-droid-fallback fonts-lato fonts-lmodern fonts-noto-mono
  fonts-texgyre fonts-urw-base35 libapache-pom-java libcommons-logging-java
  libcommons-parent-java libfontbox-java libfontenc1 libgs9 libgs9-common
  libidn12 libijs-0.35 libjbig2dec0 libkpathsea6 libpdfbox-java libptexenc1
  libruby3.0 libsynctex2 libteckit0 libtexlua53 libtexluajit2 libwoff1
  libzzip-0-13 lmodern poppler-data preview-latex-style rake ruby
  ruby-net-telnet ruby-rubygems ruby-webrick ruby-xmlrpc ruby3.0
  rubygems-integration t1utils teckit tex-common tex-gyre texlive-base
```