Visualizing dyad

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/MyDrive/smm/homeworks/utils/')
import numpy as np
from skimage import data
import matplotlib.pyplot as plt
from ImagePlot import ImagePlot
plotter = ImagePlot()
```

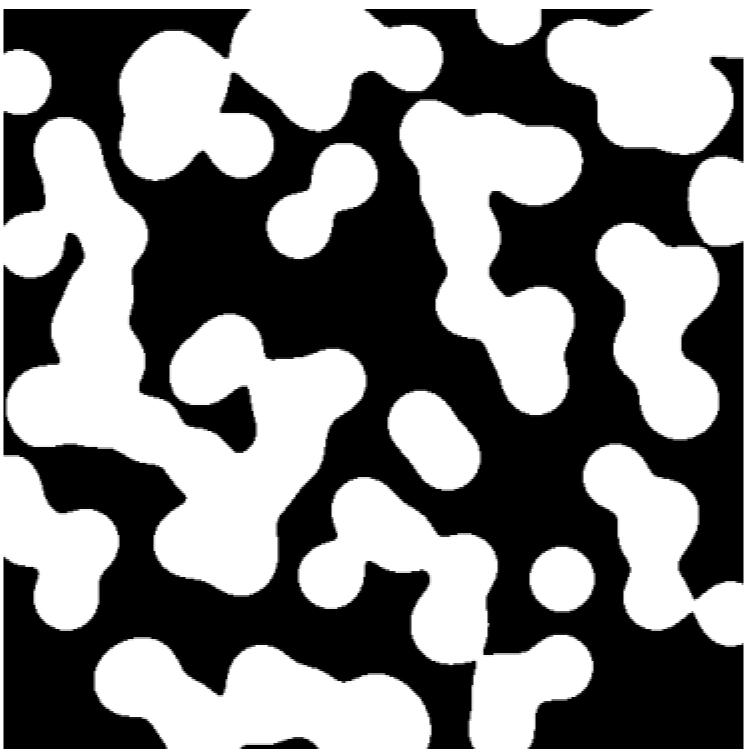
```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.moun
t("/content/drive", force_remount=True).
```

- Load the image into memory and compute its SVD;

In [ ]:
```
X = data.binary_blobs()
m, n = X.shape
print(f"Shape: {m} x {n}")
```

```
Shape: 512 x 512
```

In [ ]:
```
plt.imshow(X, cmap="gray")
plt.axis("off")
plt.show()
```

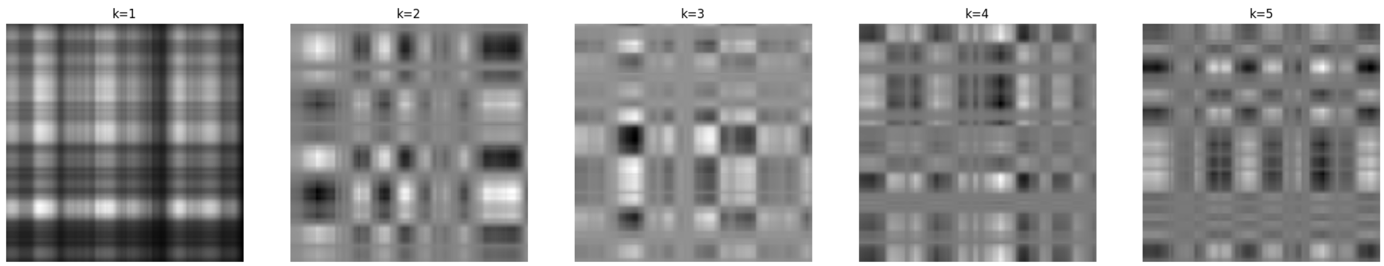


In [ ]:
```
U, s, VT = np.linalg.svd(X)
S = np.zeros(X.shape)
S[:len(s), :len(s)] = np.diag(s)
```

- Visualize some of the dyad of this decomposition.

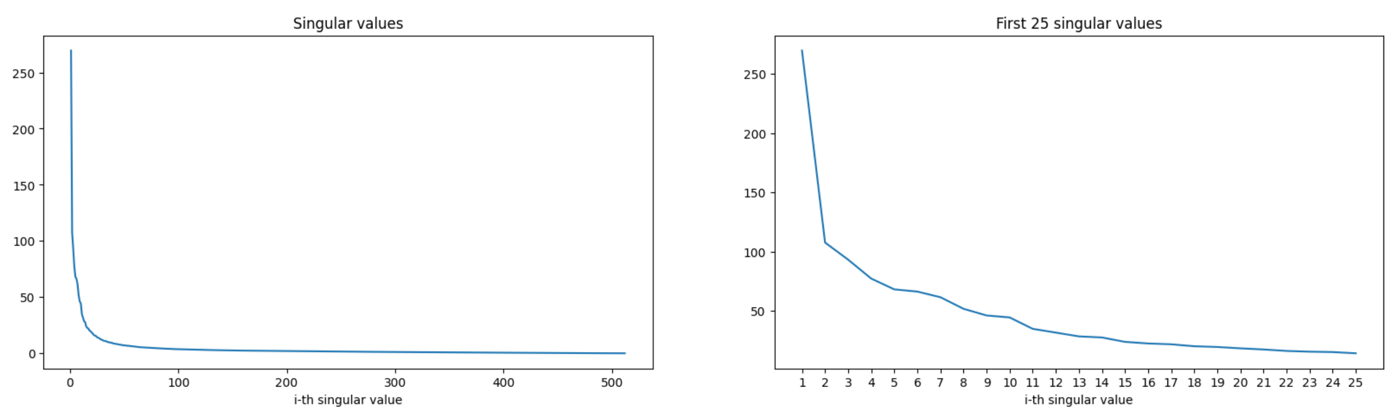
In [ ]:
```
plotter.reset()
```

```python
for k in range(1, 6):
    i = k - 1
    ui = U[:, i]
    vi = VT[i, :]
    X_k = s[i] * np.outer(ui, vi)
    #we need to generate a matrix with one row and one column, so we use the outer produ
    #outer product: we produce a matrix where each element of the first vector is multip
    plotter.add(X_k, f"k={k}")

plotter.show()
```



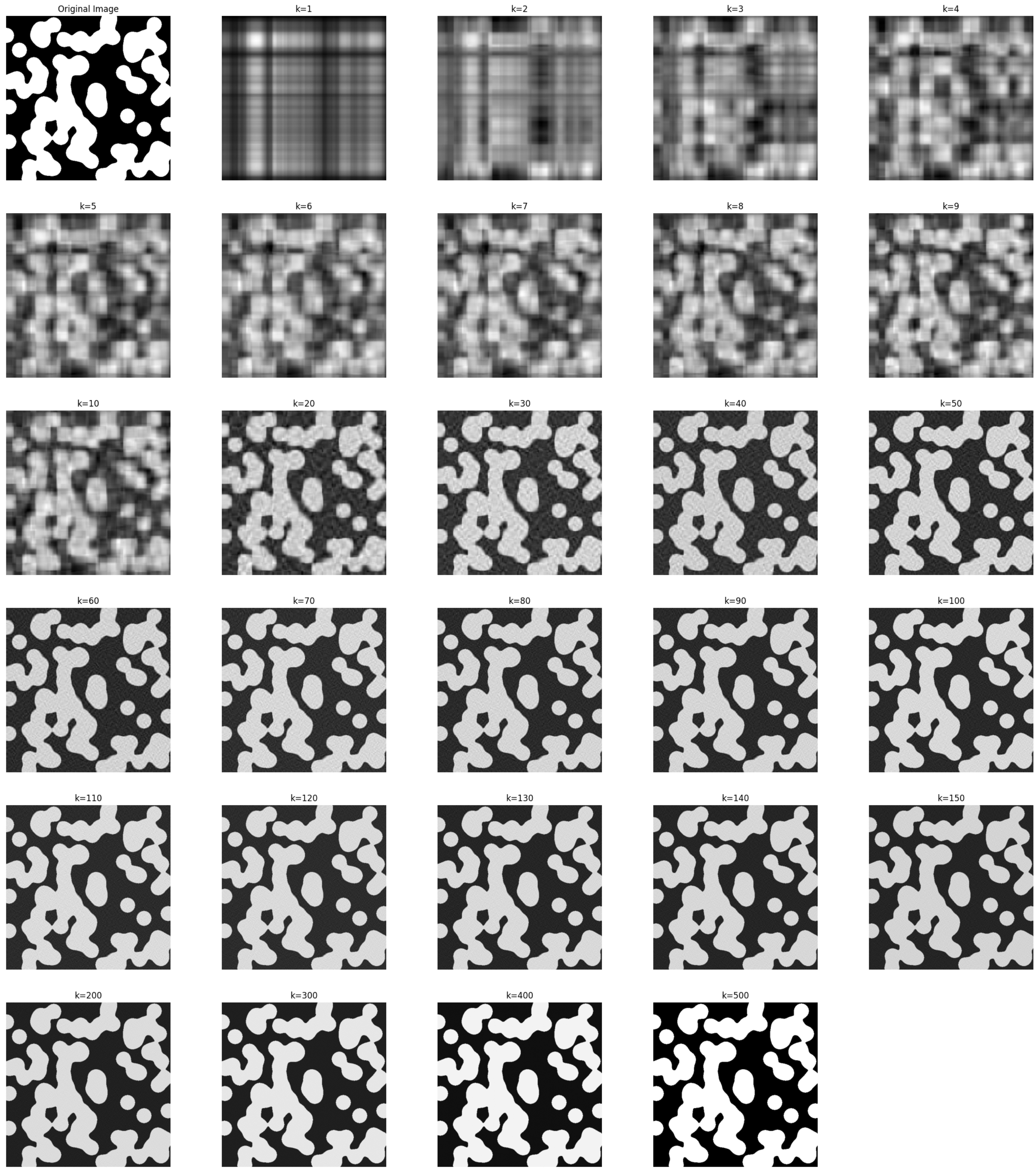
• Plot the singular values of X.

In [ ]:
```python
plt.figure(figsize=(20, 5))
plt.subplot(1, 2, 1)
plt.plot([i+1 for i in range(len(s))], s)
plt.title("Singular values")
plt.xlabel("i-th singular value")

x_limit = 25
plt.subplot(1, 2, 2)
plt.plot([i+1 for i in range(len(s[:x_limit]))], s[:x_limit])
plt.title(f"First {x_limit} singular values")
plt.xlabel("i-th singular value")
plt.xticks(ticks=[i+1 for i in range(len(s[:x_limit]))])
plt.show()
```



• Visualize the k-rank approximation of X for different values of k.

In [ ]:
```python
plotter.reset()

plotter.add(X, title="Original Image")
for k in [*range(1, 11)] + [*range(20, 151, 10)] + [*range(200, 501, 100)]:
    X_k_approx = U[:, :k] @ S[:k, :k] @ VT[:k, :]
    plotter.add(X_k_approx, title=f"k={k}")

plotter.show(figsize=(27, 30))
```



- Compute and plot the approximation error ||X − Xk||F for increasing values of k, where Xk is the k-rank approximation of k.

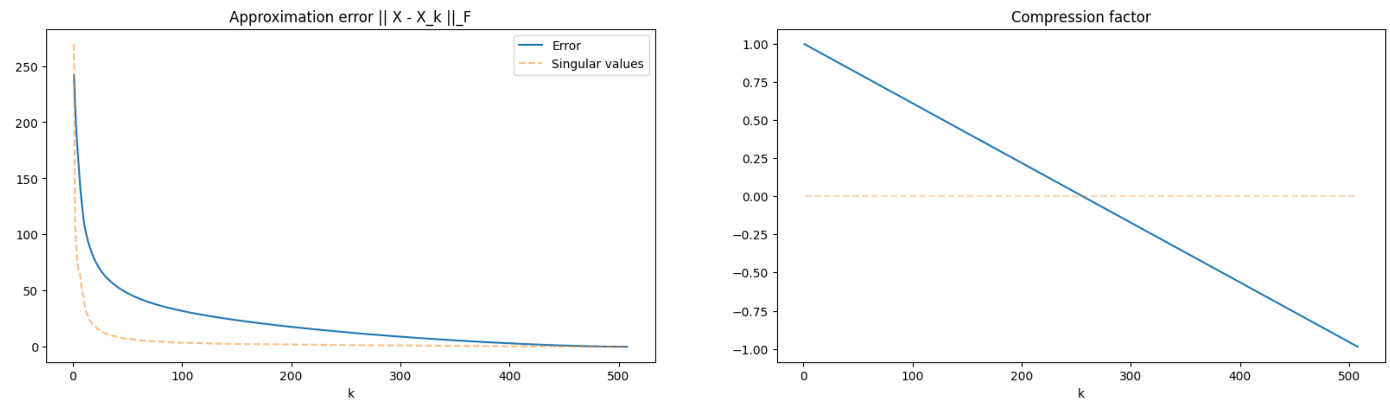
- Plot the compression factor ck = k(m+n+1)/mn for increasing k.

In [ ]:

```
approx_errors = []
compression_factors = []
to_try_k = range(1, np.linalg.matrix_rank(X)+1)
print(f'Rank of X = {np.linalg.matrix_rank(X)}')

for k in to_try_k:
    X_k_approx = U[:, :k] @ S[:k, :k] @ VT[:k, :]
    approx_errors.append( np.linalg.norm(X - X_k_approx, "fro") )
    compression_factors.append( 1 - (k*(m + n + 1)) / (m*n) ) #compression factor = valu
```

```
    if np.around(compression_factors[-1], 2) == 0:
        #print(f"Approximation error when c_k ≈ 0: {approx_errors[-1]} (k={k}) | Relativ
        print(f"Approximation error when c_k ≈ 0: {approx_errors[-1]} (k={k})")


plt.figure(figsize=(20, 5))

plt.subplot(1, 2, 1)
plt.plot(to_try_k, approx_errors, label="Error")
plt.plot(to_try_k, s[:len(to_try_k)], "--", label="Singular values", alpha=0.5)
plt.title("Approximation error || X - X_k ||_F")
plt.xlabel("k")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(to_try_k, compression_factors)
plt.plot(to_try_k, [0 for _ in to_try_k], "--", alpha=0.3)
plt.title("Compression factor")
plt.xlabel("k")
plt.show()

"""
note: compression factor = number of values that we use to represent the approximate ima
could happen that if we want to perfectly represent the original image using svd, so if 
dyads equal to the rank, we could use more values then the original image. it happens in
why we have a negative compression factor
"""
```

```
Rank of X = 508
Approximation error when c_k ≈ 0: 12.526441821974151 (k=255)
Approximation error when c_k ≈ 0: 12.441337713219271 (k=256)
Approximation error when c_k ≈ 0: 12.357246132358982 (k=257)
```



Out[ ]:

```
'\nnote: compression factor = number of values that we use to represent the approximate
image using svd\ncould happen that if we want to perfectly represent the original image
using svd, so if we take the number of\ndyads equal to the rank, we could use more value
s then the original image. it happens in that case, this is the reason\nwhy we have a ne
gative compression factor\n'
```

Classification of MNIST Digits with SVD Decomposition

In [ ]:

```
import random
import pandas as pd
from tqdm import tqdm
from itertools import combinations
random.seed(42)
import numpy as np
#np.random.seed(42)
```

Step 1 (Binary Classification)

• Load the MNIST dataset contained in ./data/MNIST.mat with the function scipy.io.loadmat.

In [ ]:

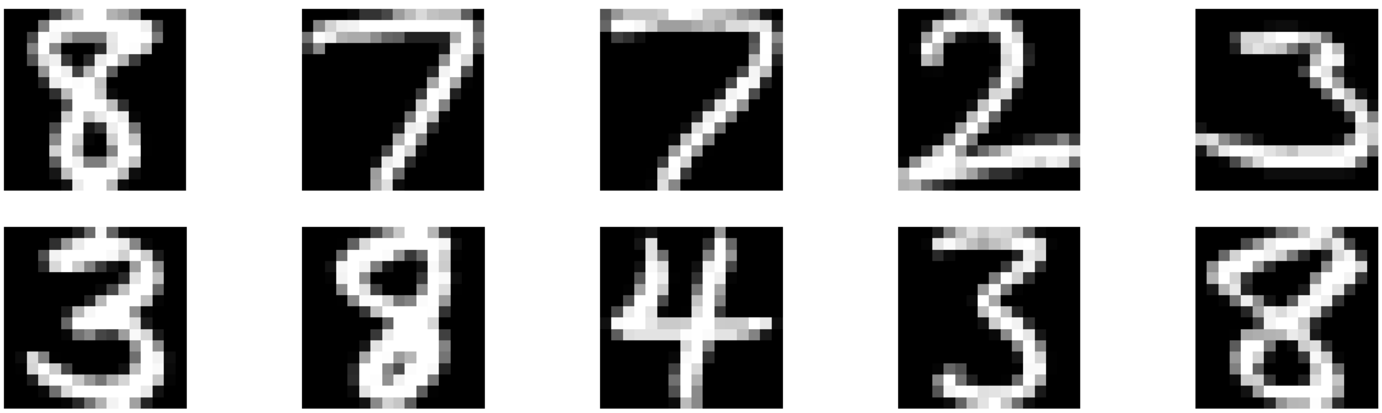
```
#data = scipy.io.loadmat("./data/MNIST.mat")
from scipy.io import loadmat
from google.colab import drive
drive.mount('/content/drive')
file_path = '/content/drive/MyDrive/smm/homeworks/homework2/data/MNIST.mat'
data = loadmat(file_path)

X = data["X"]
Y = np.squeeze(data["I"], axis=0) # if we use Y = data["I"] then we deal with the tuple

print(f"Images shape: {X.shape}")
print(f"Labels shape: {Y.shape}")
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.moun
t("/content/drive", force_remount=True).
Images shape: (256, 1707)
Labels shape: (1707,)
```

• Visualize a bunch of datapoints of X with the function plt.imshow.

In [ ]:

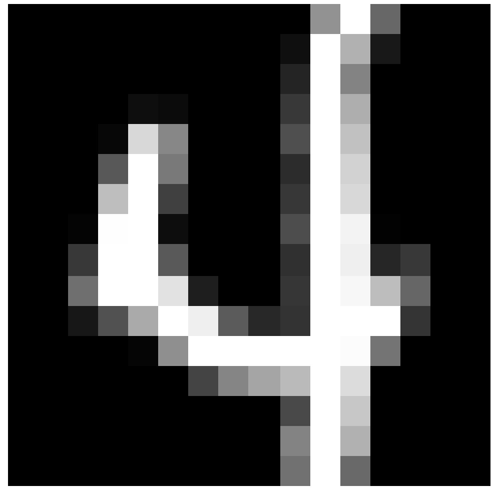
```
plotter.reset()
for i in [random.randint(0, X.shape[1]) for _ in range(10)]:
    plotter.add(X[:, i].reshape((16, 16)))
plotter.show()
```



• Extract from X those columns that corresponds to digits 3 or 4.

In [ ]:

```
def getImagesOfDigit(digit, X, Y):
    return X[:, (Y == digit)]

c1 = 3
c2 = 4
X_c1 = getImagesOfDigit(c1, X, Y)
X_c2 = getImagesOfDigit(c2, X, Y)

plotter.reset()
plotter.add(X_c1[:, 0].reshape((16, 16)))
plotter.add(X_c2[:, 0].reshape((16, 16)))
plotter.show(figsize = (18,18))
```



• Split the obtained dataset in training and testing. From now on, we will only consider the training set. The test set will be only used at the end of the exercise to test the algorithm.

In [ ]:
```
def split_train_test(X, train_ratio, random_seed=42):
    train_size = int(train_ratio * X.shape[1])
    idxs = np.arange(0, X.shape[1])
    #np.random.default_rng(random_seed).shuffle(idxs) #shuffle the data
    np.random.shuffle(idxs)
    return X[:, idxs[:train_size]], X[:, idxs[train_size:]]
```

In [ ]:
```
X_c1_train, X_c1_test = split_train_test(X_c1, 0.70)
X_c2_train, X_c2_test = split_train_test(X_c2, 0.70)
print(f"X_c1: train = {X_c1_train.shape[1]}, test = {X_c1_test.shape[1]}")
print(f"X_c2: train = {X_c2_train.shape[1]}, test: {X_c2_test.shape[1]}")
```

```
X_c1: train = 91, test = 40
X_c2: train = 85, test: 37
```

• Compute the SVD decomposition of X1 and X2 with np.linalg.svd(matrix, full matrices=False) and denote the U-part of the two decompositions as U1 and U2.

• Take an unknown digit y from the test set, and compute y1 = U1(U T 1 *y) and y 2 = U2(U T 2*y).

• Compute the distances d1 = ||y − y 1 ||2 and d2 = ||y − y 2 ||2 and classify y to C1 if d1 < d2 and to C2 if d2 < d1.

In [ ]:
```
#to obtain U matrix using SVD
def get_U(train_set_c1, train_set_c2):
    U1, _, _ = np.linalg.svd(train_set_c1, full_matrices=False)
    U2, _, _ = np.linalg.svd(train_set_c2, full_matrices=False)
    return U1, U2


def classification(digit, U1, U2, label_c1, label_c2):
    #to define the projections. digit represents the flatten matrix of each number.
    y1_projection = U1 @ (U1.T @ digit)
    y2_projection = U2 @ (U2.T @ digit)
# prima proietto nello spazio delle righe, poi riporto nello spazio originale. se proiet

    #to define the distances between the column of the trainset and the its projections.
    dist1 = np.linalg.norm(digit - y1_projection, 2)
    dist2 = np.linalg.norm(digit - y2_projection, 2)

    #check to define the label
    if dist1 < dist2:
```

```
        return label_c1
    else:
        return label_c2

def evaluation(U1, U2, c1, X_c1, c2, X_c2):
    correct_c1 = sum(classification(X_c1[:, i], U1, U2, c1, c2) == c1 for i in range(X_c
    correct_c2 = sum(classification(X_c2[:, i], U1, U2, c1, c2) == c2 for i in range(X_c
    return correct_c1, correct_c2

def BinaryClassifier(c1, X_c1_train, X_c1_test, c2, X_c2_train, X_c2_test):
    U1, U2 = get_U(X_c1_train, X_c2_train)
    c1_correct_train, c2_correct_train = evaluation(U1, U2, c1, X_c1_train, c2, X_c2_tra
    c1_correct_test, c2_correct_test = evaluation(U1, U2, c1, X_c1_test, c2, X_c2_test)

    return {
        "train": {"correct1": c1_correct_train, "correct2": c2_correct_train},
        "test": {"correct1": c1_correct_test, "correct2": c2_correct_test}
    }

results = BinaryClassifier(3, X_c1_train, X_c1_test, 4, X_c2_train, X_c2_test)
print(f'train data = 70% ({X_c1_train.shape[1]+X_c2_train.shape[1]}), test data = 30% ({
print(f"Accuracy train (3): {results['train']['correct1'] / X_c1_train.shape[1]:.3f} ({X
print(f"Accuracy train (4): {results['train']['correct2'] / X_c2_train.shape[1]:.3f} ({X
print(f"Accuracy test (3): {results['test']['correct1'] / X_c1_test.shape[1]:.3f} ({X_c1
print(f"Accuracy test (4): {results['test']['correct2'] / X_c2_test.shape[1]:.3f} ({X_c2
```

```
train data = 70% (176), test data = 30% (77)
Accuracy train (3): 1.000 (0/91 wrong)
Accuracy train (4): 1.000 (0/85 wrong)
Accuracy test (3): 1.000 (0/40 wrong)
Accuracy test (4): 1.000 (0/37 wrong)
```

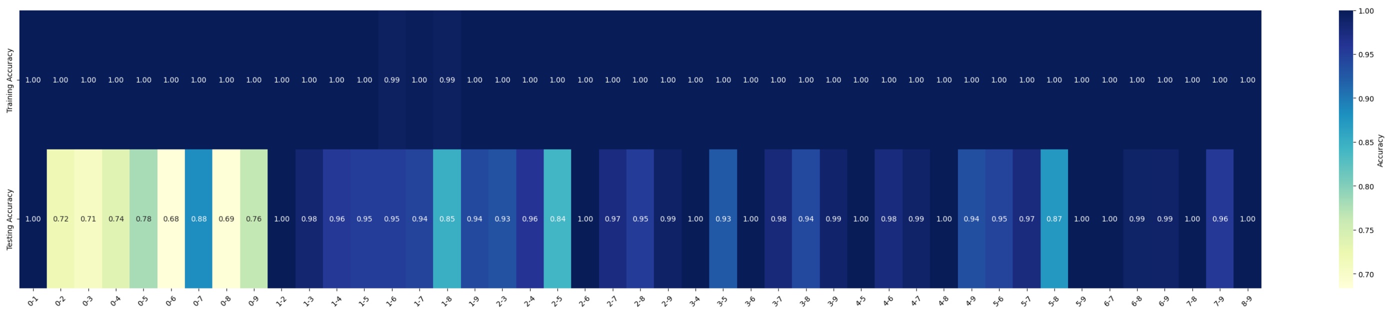
- Repeat the experiment for different values of y in the test set. Compute the misclassification number for this algorithm.

In [ ]:
```
X_c1_train, X_c1_test = split_train_test(X_c1, 0.50)
X_c2_train, X_c2_test = split_train_test(X_c2, 0.50)
print(f'train data = 50% ({X_c1_train.shape[1]+X_c2_train.shape[1]}), test data = 50% ({
results = BinaryClassifier(3, X_c1_train, X_c1_test, 4, X_c2_train, X_c2_test)
print(f"Accuracy train (3): {results['train']['correct1'] / X_c1_train.shape[1]:.3f} ({X
print(f"Accuracy train (4): {results['train']['correct2'] / X_c2_train.shape[1]:.3f} ({X
print(f"Accuracy test (3): {results['test']['correct1'] / X_c1_test.shape[1]:.3f} ({X_c1
print(f"Accuracy test (4): {results['test']['correct2'] / X_c2_test.shape[1]:.3f} ({X_c2
X_c1_train, X_c1_test = split_train_test(X_c1, 0.60)
X_c2_train, X_c2_test = split_train_test(X_c2, 0.60)
print(f'train data = 60% ({X_c1_train.shape[1]+X_c2_train.shape[1]}), test data = 40% ({
results = BinaryClassifier(3, X_c1_train, X_c1_test, 4, X_c2_train, X_c2_test)
print(f"Accuracy train (3): {results['train']['correct1'] / X_c1_train.shape[1]:.3f} ({X
print(f"Accuracy train (4): {results['train']['correct2'] / X_c2_train.shape[1]:.3f} ({X
print(f"Accuracy test (3): {results['test']['correct1'] / X_c1_test.shape[1]:.3f} ({X_c1
print(f"Accuracy test (4): {results['test']['correct2'] / X_c2_test.shape[1]:.3f} ({X_c2
```

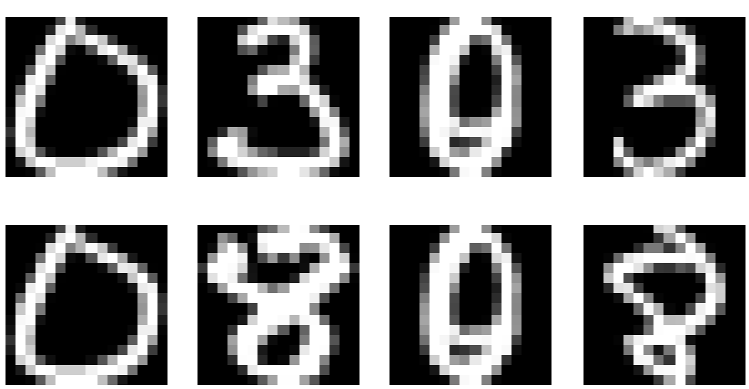
```
train data = 50% (126), test data = 50% (127)
Accuracy train (3): 1.000 (0/65 wrong)
Accuracy train (4): 1.000 (0/61 wrong)
Accuracy test (3): 1.000 (0/66 wrong)
Accuracy test (4): 0.984 (1/61 wrong)
train data = 60% (151), test data = 40% (102)
Accuracy train (3): 1.000 (0/78 wrong)
Accuracy train (4): 1.000 (0/73 wrong)
Accuracy test (3): 1.000 (0/53 wrong)
Accuracy test (4): 1.000 (0/49 wrong)
```

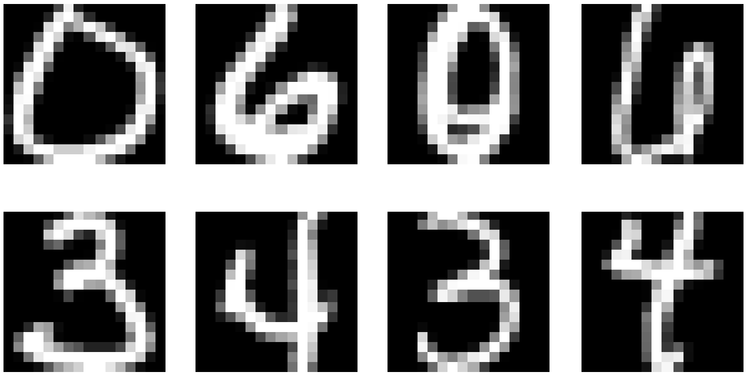
- Repeat the experiment for different digits other than 3 or 4.

In [ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

results_matrix = np.zeros((2, 45)) # 45 = all possible couple of digits
column_counter = 0
for i in range(0, 10):
    for j in range(i + 1, 10):  # Only distinct pairs
        X_c1 = getImagesOfDigit(i, X, Y)
        X_c2 = getImagesOfDigit(j, X, Y)
        X_c1_train, X_c1_test = split_train_test(X_c1, 0.70)
        X_c2_train, X_c2_test = split_train_test(X_c2, 0.70)
        results = BinaryClassifier(i, X_c1_train, X_c1_test, j, X_c2_train, X_c2_test)
        accuracy_train = (results['train']['correct1'] + results['train']['correct2']) /
        accuracy_test = (results['test']['correct1'] + results['test']['correct2']) / (X
        # Store the accuracies in the matrix
        results_matrix[0, column_counter] = accuracy_train
        results_matrix[1, column_counter] = accuracy_test
        column_counter += 1
column_labels = [f"{i}-{j}" for i in range(0, 10) for j in range(i + 1, 10)]
row_labels = ['Training Accuracy', 'Testing Accuracy']

plt.figure(figsize=(30, 6))
sns.heatmap(results_matrix, annot=True, fmt='.2f', cmap="YlGnBu", xticklabels=column_lab
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



In [ ]:
```python
digit_pairs = [(0, 3),(0, 8),(0, 6),(3, 4)]
for c1, c2 in digit_pairs:
    X_c1 = getImagesOfDigit(c1, X, Y)
    X_c2 = getImagesOfDigit(c2, X, Y)
    plotter.reset()
    for _ in range(2):
        plotter.add(X_c1[:, _].reshape((16, 16)))
        plotter.add(X_c2[:, _].reshape((16, 16)))
    plotter.show(figsize=(6, 6))
```





Step 2 (Multiclass classification)

The extension of this idea to the multiple classification task is trivial. Indeed, if we have more than 2 classes (say, k different classes) C1, . . . , Ck, we just need to repeat the same procedure as before for each matrix X1, . . . , Xk to obtain the distances d1, . . . , dk. Then, the new digit y will be classified as Ci if di is lower that dj for each j = 1, . . . , k.

In [ ]:

```
import numpy as np

# U matrix for each possible digit
def get_U_multiclass(train_sets):
    Us = []
    for train_set in train_sets:
        U, _, _ = np.linalg.svd(train_set, full_matrices=False)
        Us.append(U)
    return Us

def classification_multiclass(digit, Us, labels):
    min_dist = float('inf')
    best_label = None

    for U, label in zip(Us, labels):
        projection = U @ (U.T @ digit)
        dist = np.linalg.norm(digit - projection, 2)
        # if the distance is lower then the previous one I update the result
        if dist < min_dist:
            min_dist = dist
            best_label = label
    return best_label

def evaluation_multiclass(Us, labels, test_sets):
    correct_counts = {label: 0 for label in labels} #now i use a dictionary to solve all
    for label, X in zip(labels, test_sets):
        correct_counts[label] = sum(classification_multiclass(X[:, i], Us, labels) == la
    return correct_counts

def MultiClassClassifier(labels, train_sets, test_sets):
    Us = get_U_multiclass(train_sets)
    correct_counts_train = evaluation_multiclass(Us, labels, train_sets)
    correct_counts_test = evaluation_multiclass(Us, labels, test_sets)

    total_counts_train = {label: X.shape[1] for label, X in zip(labels, train_sets)}
    total_counts_test = {label: X.shape[1] for label, X in zip(labels, test_sets)}

    # dictionaty where key train contains the dictionary of train results and the same f
    results = {
        "train": {label: correct_counts_train[label] / total_counts_train[label] for lab
        "test": {label: correct_counts_test[label] / total_counts_test[label] for label
```

```
    }
    return results
```

Repeat the exercise above with a 3-digit example.

In [ ]:
```
import random
labels = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in range(5):
    selected_labels = random.sample(labels, 3)
    print(f"Labels {selected_labels}")

    train_sets = []
    test_sets = []
    for digit in selected_labels:
        train, test = split_train_test(X[:, (Y == digit)], 0.70)
        train_sets.append(train)
        test_sets.append(test)

    results = MultiClassClassifier(selected_labels, train_sets, test_sets)

    for label in selected_labels:
        print(f"Accuracy train ({label}): {results['train'][label]:.3f} ({train_sets[sel
        print(f"Accuracy test ({label}): {results['test'][label]:.3f} ({test_sets[select
    print("\n")
```

```
Labels [3, 5, 1]
Accuracy train (3): 1.000 (0/91 wrong)
Accuracy test (3): 0.925 (3/40 wrong)
Accuracy train (5): 1.000 (0/61 wrong)
Accuracy test (5): 0.519 (13/27 wrong)
Accuracy train (1): 1.000 (0/176 wrong)
Accuracy test (1): 1.000 (0/76 wrong)


Labels [1, 6, 9]
Accuracy train (1): 1.000 (0/176 wrong)
Accuracy test (1): 0.987 (1/76 wrong)
Accuracy train (6): 0.990 (1/105 wrong)
Accuracy test (6): 0.957 (2/46 wrong)
Accuracy train (9): 1.000 (0/92 wrong)
Accuracy test (9): 0.775 (9/40 wrong)


Labels [5, 9, 4]
Accuracy train (5): 1.000 (0/61 wrong)
Accuracy test (5): 0.852 (4/27 wrong)
Accuracy train (9): 1.000 (0/92 wrong)
Accuracy test (9): 1.000 (0/40 wrong)
Accuracy train (4): 1.000 (0/85 wrong)
Accuracy test (4): 0.838 (6/37 wrong)


Labels [0, 7, 1]
Accuracy train (0): 1.000 (0/223 wrong)
Accuracy test (0): 1.000 (0/96 wrong)
Accuracy train (7): 1.000 (0/116 wrong)
Accuracy test (7): 0.460 (27/50 wrong)
Accuracy train (1): 1.000 (0/176 wrong)
Accuracy test (1): 1.000 (0/76 wrong)


Labels [6, 1, 4]
Accuracy train (6): 0.981 (2/105 wrong)
Accuracy test (6): 0.957 (2/46 wrong)
Accuracy train (1): 1.000 (0/176 wrong)
```

```
Accuracy test (1): 0.987 (1/76 wrong)
Accuracy train (4): 0.988 (1/85 wrong)
Accuracy test (4): 0.703 (11/37 wrong)
```

the worst result are given by the digits that have the lowest number of sample in the training set. It's important defining as general as possible row space in order to be able to classify as much as possible different version of the same digit

Clustering with PCA

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
file_path = '/content/drive/MyDrive/smm/homeworks/homework2/data/data.csv'

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tqdm import tqdm
from itertools import combinations

np.random.seed(42)
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.moun
t("/content/drive", force_remount=True).
```

• Load the dataset in memory and explore its head and shape to understand how the informations are placed inside of it;

In [ ]:
```
data = pd.read_csv(file_path)
print(f"Data shape: {data.shape}")
display(data.head())
```

```
Data shape: (42000, 785)
```

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776
0	1	0	0	0	0	0	0	0	0	0	...	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0
2	1	0	0	0	0	0	0	0	0	0	...	0	0	0
3	4	0	0	0	0	0	0	0	0	0	...	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0

5 rows × 785 columns

• Split the dataset into the X matrix of dimension d × N, with d = 784 being the dimension of each datum, N is the number of datapoints, and Y ∈ R_n containing the corresponding labels;

In [ ]:
```
data = data.to_numpy()
X_full = data[:, 1:].T
Y_full = data[:, 0]
print(X_full.shape, Y_full.shape)
```

```
(784, 42000) (42000,)
```

• Choose a number of digits (for example, 0, 6 and 9) and extract from X and Y the sub-dataset containing only the considered digits. Re-call X and Y those datasets, since the originals are not required anymore;

• Set Ntrain < N and randomly sample a training set with Ntrain datapoints from X (and the corresponding Y). Call them Xtrain and Ytrain. Everything else is the test set. Call it Xtest and Ytest.

In [ ]:
```
def extractDigits(X, Y, to_selected_digits):
    # isin -> creates the boolean mask
    mask = np.isin(Y, to_selected_digits)
    return X[:, mask], Y[mask]

def train_test_split(X, Y, train_size, random_seed=42):
    rng = np.random.default_rng(random_seed)
    idxs = np.arange(0, X.shape[1])
    rng.shuffle(idxs)
    X_train = X[:, idxs[:train_size]]
    Y_train = Y[idxs[:train_size]]
    X_test = X[:, idxs[train_size:]]
    Y_test = Y[idxs[train_size:]]
    return X_train, Y_train, X_test, Y_test
```

In [ ]:
```
X, Y = extractDigits(X_full, Y_full, [0, 6, 9])
X_train, Y_train, X_test, Y_test = train_test_split(X, Y, int(0.75 * X.shape[1]))

print(f"Train set: {X_train.shape}, {Y_train.shape}")
print(f"Test set: {X_test.shape}, {Y_test.shape}")
```

```
Train set: (784, 9342), (9342,)
Test set: (784, 3115), (3115,)
```

• Implement the algorithms computing the PCA of Xtrain with a fixed k. Visualize the results (for k = 2) and the position of the centroid of each cluster;

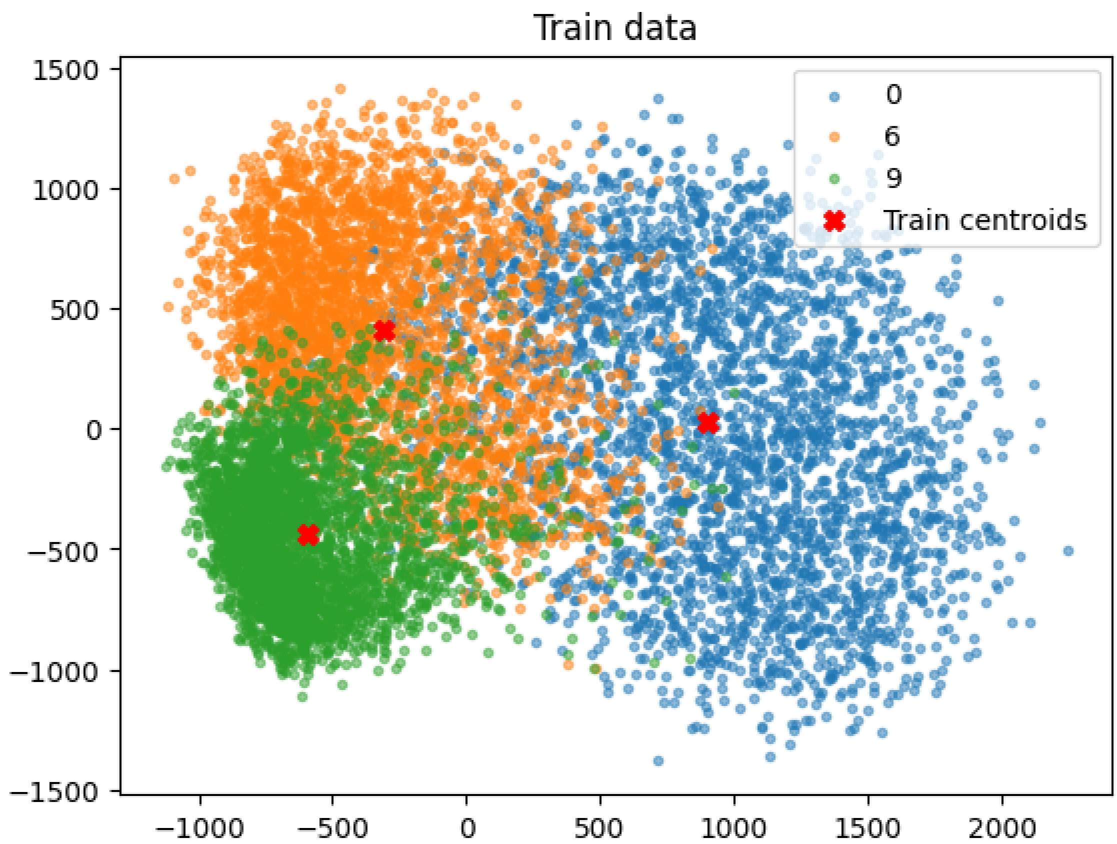
In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

def centroid(data):
    return np.expand_dims(np.mean(data, axis=1), 1) # to allow the subtraction: (m,n)-(m

def get_U(data, k):
    #subtract the centroid by each column
    data_centroid = centroid(data)
    centered_data = data - data_centroid
    #SVD
    U, _, _ = np.linalg.svd(centered_data, full_matrices=False)
    # Projection matrix using the first k components
    proj_matrix = U[:, :k].T
    return proj_matrix, data_centroid

def projection(data, proj_matrix, data_centroid):
    # If a single data point is provided, we use it in the next exercise when we evaluat
    if data.ndim == 1:
        data = np.expand_dims(data, axis=1)
    data_centered = data - data_centroid
    return proj_matrix @ data_centered

def pca(data, k):
    proj_matrix, data_centroid = get_U(data, k)
    return projection(data, proj_matrix, data_centroid), proj_matrix, data_centroid

def plot_data(Z_k, Y, train_centroids=[], test_centroids=[], title=""):
    # Data points
    for digit in np.unique(Y):
        plt.scatter(Z_k[0, Y==digit], Z_k[1, Y==digit], label=digit, marker=".", alpha=0
    # Centroids
    if len(train_centroids) > 0:
```

```
        plt.scatter([c[0, :] for c in train_centroids], [c[1, :] for c in train_centroid
                    c="red", s=50, marker="X", label="Train centroids")
    if len(test_centroids) > 0:
        plt.scatter([c[0, :] for c in test_centroids], [c[1, :] for c in test_centroids]
                    c="purple", s=50, marker="x", label="Test centroids")

    plt.title(title)
    plt.legend(scatterpoints=1)
    plt.show()
```

visualize the results

In [ ]:
```
Z_k_train, proj_matrix_train, data_centroid_train = pca(X_train, 2)
print(Z_k_train.shape)
plot_data(Z_k_train, Y_train, title="Train data", train_centroids=[centroid(Z_k_train[:,
```

```
(2, 9342)
```



- Compute, for each cluster, the average distance from the centroid.

In [ ]:
```
for digit in np.unique(Y_train):
    Z_cluster = Z_k_train[:, Y_train == digit]
    cluster_centroid = centroid(Z_cluster) # compute centroid for each cluster
    dists = []
    for i in range(Z_cluster.shape[1]): # for each cluster's value
        assert cluster_centroid[:, 0].shape == Z_cluster[:, i].shape #each value must ha
        dists.append( np.linalg.norm(cluster_centroid[:, 0] - Z_cluster[:, i], 2) ) # co
    print(f"label = {digit}: train avg. distance to centroid equal to {np.mean(dists)}")
```

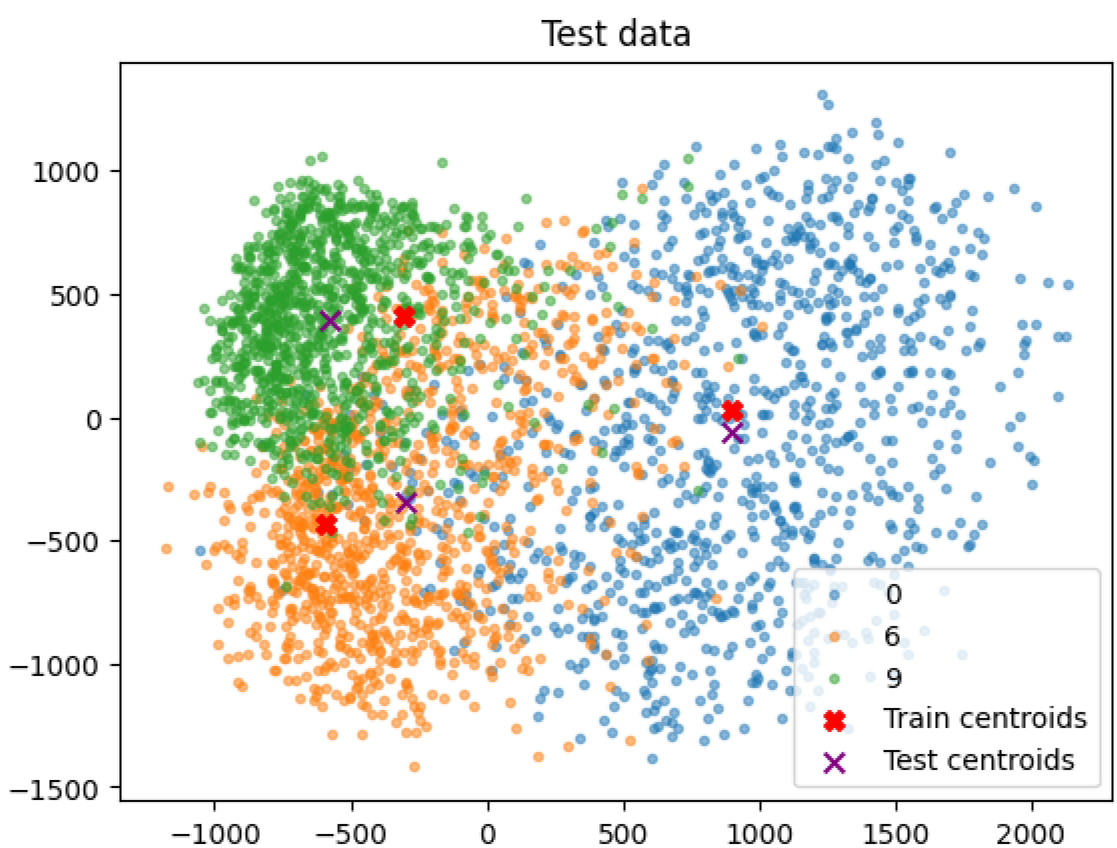
```
label = 0: train avg. distance to centroid equal to 725.8976012407383
label = 6: train avg. distance to centroid equal to 530.0432897943263
label = 9: train avg. distance to centroid equal to 356.54929356719646
```

- Compute, for each cluster, the average distance from the centroid on the test set.

In [ ]:
```
Z_k_test, proj_matrix_test, data_centroid_test = pca(X_test, 2)
print(Z_k_test.shape)
```

```
print('\nAvergare distances using the test set:')
for digit in np.unique(Y_test):
    Z_cluster = Z_k_test[:, Y_test == digit]
    cluster_centroid = centroid(Z_cluster)

    dists = []
    for i in range(Z_cluster.shape[1]):
        assert cluster_centroid[:, 0].shape == Z_cluster[:, i].shape
        dists.append( np.linalg.norm(cluster_centroid[:, 0] - Z_cluster[:, i], 2) )
    print(f"{digit} | test avg. distance to centroid {np.mean(dists)}")

print('\n')
plot_data(
    Z_k_test, Y_test, title="Test data",
    train_centroids = [centroid(Z_k_train[:, Y_train==digit]) for digit in np.unique(Y_t
    test_centroids = [centroid(Z_k_test[:, Y_test==digit]) for digit in np.unique(Y_test
)
```

```
(2, 3115)

Avergare distances using the test set:
0 | test avg. distance to centroid 751.71911312475
6 | test avg. distance to centroid 558.7475202517368
9 | test avg. distance to centroid 380.1842464913515
```



• Define a classification algorithm in this way: given a new observation x, compute the distance between x and each cluster centroid. Assign x to the class corresponding the the closer centroid. Compute the accuracy of this algorithm on the test set;

In [ ]:

```
import numpy as np

def compute_centroids(Z_k_train, Y_train):
    possible_digits = np.unique(Y_train)
    centroids = {digit: centroid(Z_k_train[:, Y_train == digit]) for digit in possible_d
    return centroids, possible_digits
```

```
def predict_digit(proj_matrix, data_centroid, centroids, possible_digits, new_digit):
    Z_k_digit = projection(new_digit, proj_matrix, data_centroid) # define the projectio
    best_distance = +np.inf
    best_digit = None

    for digit in possible_digits:
        distance = np.linalg.norm(centroids[digit] - Z_k_digit, 2) # compute the distanc
        if distance < best_distance:
            best_distance = distance
            best_digit = digit
    # at the end we make the predition, so we assign the closest centroid label to each
    return best_digit

def evaluate_model(proj_matrix, data_centroid, centroids, possible_digits, X_test, Y_tes
    correct = 0
    total_digits = X_test.shape[1] # all the possible values

    for i in range(total_digits):
        prediction = predict_digit(proj_matrix, data_centroid, centroids, possible_digit
        if prediction == Y_test[i]: # we check if the prediction corresponds to the true
            correct += 1

    accuracy = correct / total_digits
    return accuracy, correct, total_digits
```

In [ ]:
```
# Step 1: define the U matrix, the projected training set and the data centroid
Z_k_train, proj_matrix, data_centroid = pca(X_train, k=2)
# Step 2: compute the centroid for each label in the traning set
centroids, possible_digits = compute_centroids(Z_k_train, Y_train)
# Step 3: evaluation of the test set
accuracy, correct, total_digits = evaluate_model(proj_matrix, data_centroid, centroids,
print(f"Accuracy of the clustering model on the test set: {accuracy:.5f} ({correct}/{tot
```

```
Accuracy of the clustering model on the test set: 0.84751 (2640/3115 correct)
```

- Repeat this experiment for different values of k and different digits.

In [ ]:
```
# just to try different value using a for cycle
def evaluateOnDigits(digits, X_full, Y_full, k=2):
    # try using differents digits and k values
    X, Y = extractDigits(X_full, Y_full, digits)
    X_train, Y_train, X_test, Y_test = train_test_split(X, Y, int(0.75 * X.shape[1]), ra

    Z_k_train, proj_matrix, data_centroid = pca(X_train, k)
    centroids, possible_digits = compute_centroids(Z_k_train, Y_train)
    accuracy, correct, total_digits = evaluate_model(proj_matrix, data_centroid, centroi
    return accuracy
```

In [ ]:
```
def evaluate_multiple_runs(X_full, Y_full, num_runs=5, k_values=[2,3,5,15]):
    results = []
    for i in range(num_runs):
        digits = np.random.choice(range(10), 3, replace=False)
        print(f"Testing on digits {digits}")
        for k in k_values:
            accuracy = evaluateOnDigits(digits, X_full, Y_full, k)
            results.append((digits, k, accuracy))
            print(f"  k={k}, Accuracy: {accuracy:.5f}")
    return results

results = evaluate_multiple_runs(X_full, Y_full)
```

```
Testing on digits [8 1 5]
  k=2, Accuracy: 0.74809
```

```
  k=3, Accuracy: 0.78316
  k=5, Accuracy: 0.87022
  k=15, Accuracy: 0.89190
Testing on digits [0 1 8]
  k=2, Accuracy: 0.91677
  k=3, Accuracy: 0.92888
  k=5, Accuracy: 0.93789
  k=15, Accuracy: 0.94068
Testing on digits [9 2 0]
  k=2, Accuracy: 0.92960
  k=3, Accuracy: 0.93344
  k=5, Accuracy: 0.93792
  k=15, Accuracy: 0.94336
Testing on digits [1 7 6]
  k=2, Accuracy: 0.94495
  k=3, Accuracy: 0.94949
  k=5, Accuracy: 0.95100
  k=15, Accuracy: 0.95554
Testing on digits [1 5 4]
  k=2, Accuracy: 0.87604
  k=3, Accuracy: 0.89388
  k=5, Accuracy: 0.90854
  k=15, Accuracy: 0.92957
```