hw5-programming-worked-out

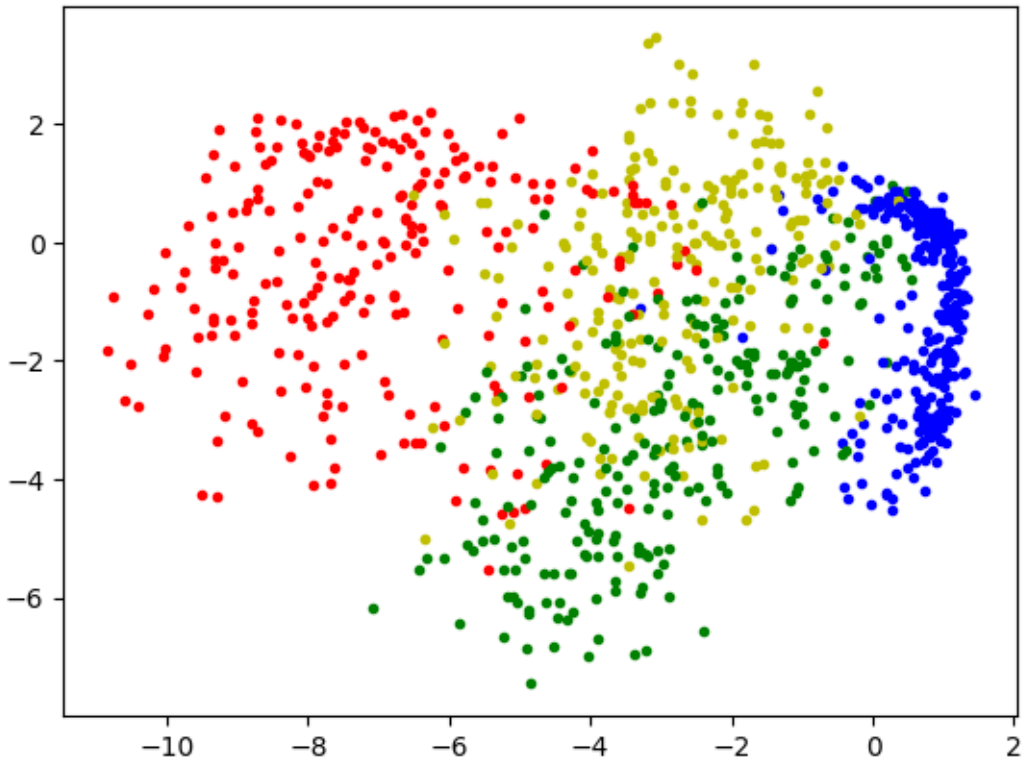
October 29, 2024

[ ]:
```
#do not change
import sys
import os
from time import time
%matplotlib inline
from urllib.request import urlopen
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import numpy as np
#from scipy.misc import imsave
import math

def load_mnist():
    images_url = 'https://github.com/guptashvm/Data/blob/master/data/
 ↪train-images-idx3-ubyte?raw=true'
    with urlopen(images_url) as urlopened:
      fd = urlopened.read()
      loaded = np.frombuffer(fd,dtype=np.uint8)
      trX = loaded[16:].reshape((60000,28*28)).astype(float)

    labels_url = 'https://github.com/guptashvm/Data/blob/master/data/
 ↪train-labels-idx1-ubyte?raw=true'
    with urlopen(labels_url) as urlopened:
      fd = urlopened.read()
      loaded = np.frombuffer(fd,dtype=np.uint8)
      trY = loaded[8:].reshape((60000))

    trY = np.asarray(trY)

    X = trX / 255.
    y = trY

    subset  = [i for i, t in enumerate(y) if t in [1, 0, 2, 3]]
    X, y = X.astype('float32')[subset], y[subset]
    return X[:1000], y[:1000]
```

Run the following code to load the data we need. Here, **X** is the array of images, **y** is the array of labels, and **X2d** is the array of projections of the images into two-dimensional space using PCA.

The two-dimensional scatterplot of the top two principal components of the images is displayed, where the color of each point represents its label.

[ ]:
```
X, y = load_mnist()
pca = PCA(n_components=2)
pca.fit(X)
X2d = X.dot(pca.components_.T)

def plot_with_colors(Xs, ys):
  for i, _ in enumerate(ys):
    if ys[i] == 0:
      plt.plot([Xs[i, 0]], [Xs[i, 1]], 'r.')
    elif ys[i] == 1:
      plt.plot([Xs[i, 0]], [Xs[i, 1]], 'b.')
    elif ys[i] == 2:
      plt.plot([Xs[i, 0]], [Xs[i, 1]], 'g.')
    elif ys[i] == 3:
      plt.plot([Xs[i, 0]], [Xs[i, 1]], 'y.')
  plt.show()

plot_with_colors(X2d, y)
```



(a) Implement the standard k-means algorithm. Please complete the function **kmeans** defined

below. You are NOT allowed to use any existing code of **kmeans** for this problem.

```
[ ]: def kmeans(X, k = 4, max_iter = 500, random_state=0):
       """
       Inputs:
           X: input data matrix, numpy array with shape (n * d), n: number of data␣
     ↪points, d: feature dimension
           k: number of clusters
           max_iters: maximum iterations
       Output:
           clustering label for each data point
       """
       assert len(X) > k, 'illegal inputs'
       np.random.seed(random_state)

       # randomly select k data points as centers
       idx = np.random.choice(len(X), k, replace=False)
       centers = X[idx]

       # please complete the following code:

       from scipy.spatial import distance
       for i in range(max_iter):
           H = distance.cdist(X, centers, 'euclidean')
           # update clustering labels
           labels = np.argmin(H, axis=1)
           # update new centers
           new_centers = []
           for c in range(k):
               # select points belonging to cluster c
               cluster_points = X[labels == c]

               # check if the cluster has any points assigned
               if len(cluster_points) > 0:
                   # calculate the mean of points in cluster c and add to new_centers
                   new_centers.append(cluster_points.mean(axis=0))
               else:
                   # if no points are assigned to cluster c, keep the previous center
                   new_centers.append(centers[c])

           # convert list to numpy array
           new_centers = np.array(new_centers)

           #check for convergence
           if np.allclose(centers, new_centers, atol=1e-6):
               break
           # update the centers
```

```
        centers = new_centers

    return labels
```

(b) Run your **kmeans** function on the dataset (of the top two PCA components given by array X2d). Set the number of clusters to 4. Visualize the result by coloring the 2D points in (a) according to their **clustering labels** returned by your **kmeans** algorithm. Because **kmeans** is sensitive to initialization, repeat your **kmeans** at least 5 times with different random initializations and show the plot of each initialization.

To quantitatively evaluate the clustering performance, we evaluate the *unsupervised clustering accuracy*, which can be written as follows,

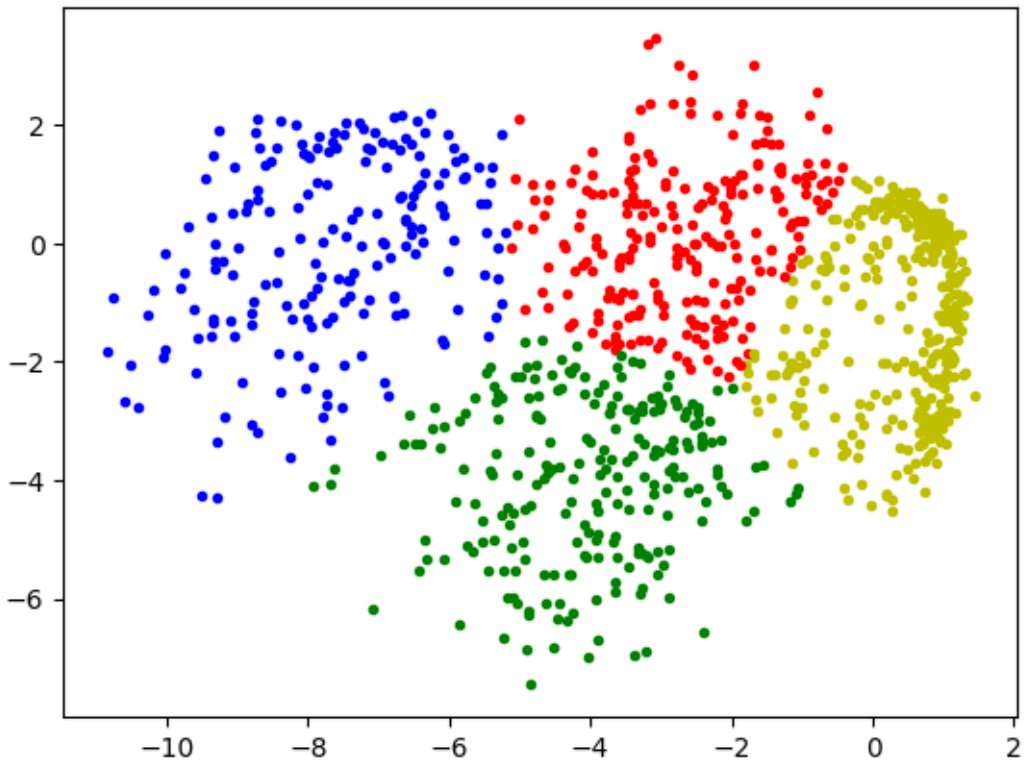
$$\text{accuracy} = \max_{\mathcal{M}} \frac{\sum_{i=1}^{n} \mathbb{I}(y_i = \mathcal{M}(z_i))}{n}, n = 1000,$$

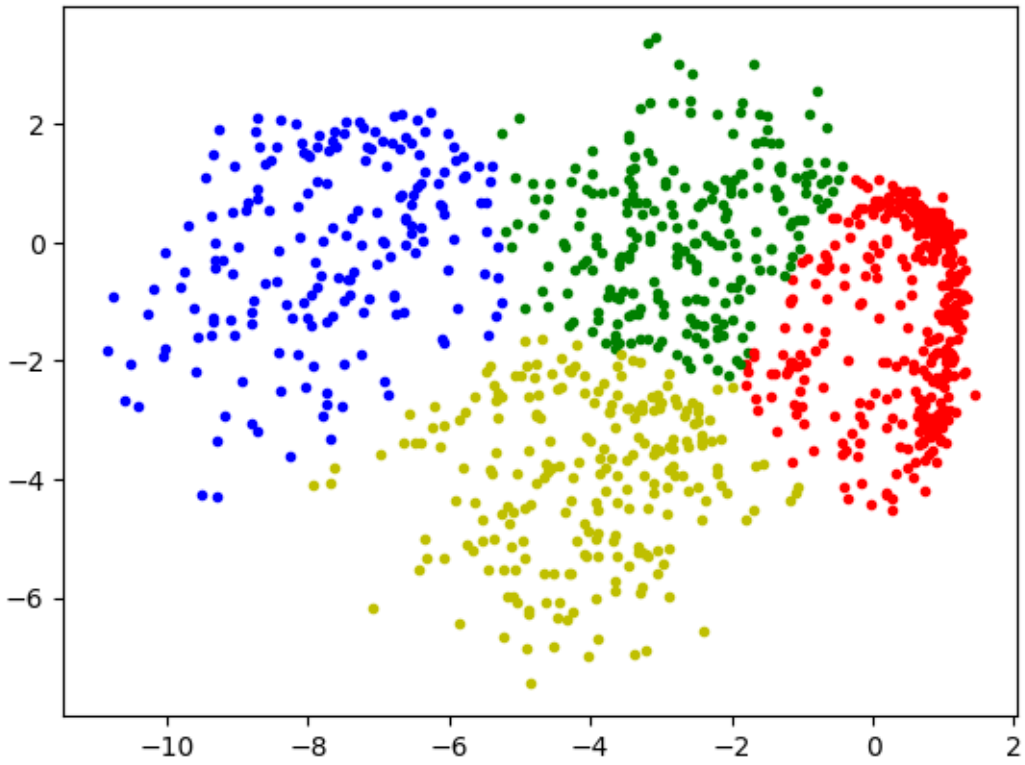
where y_i is the ground-truth label, z_i is the cluster assignment produced by the algorithm, and $\mathcal{M}$ ranges over all possible one-to-one mapping between clusters and labels and $\mathbb{I}(x)$ is a indicator function ($\mathbb{I}(x) = 1 \text{if } x = 1; \text{otherwise } 0$). Please use the **accuracy_score** function defined below to calculate the accuracy. Report the best clustering accuracy you get out of 10 random initializations.

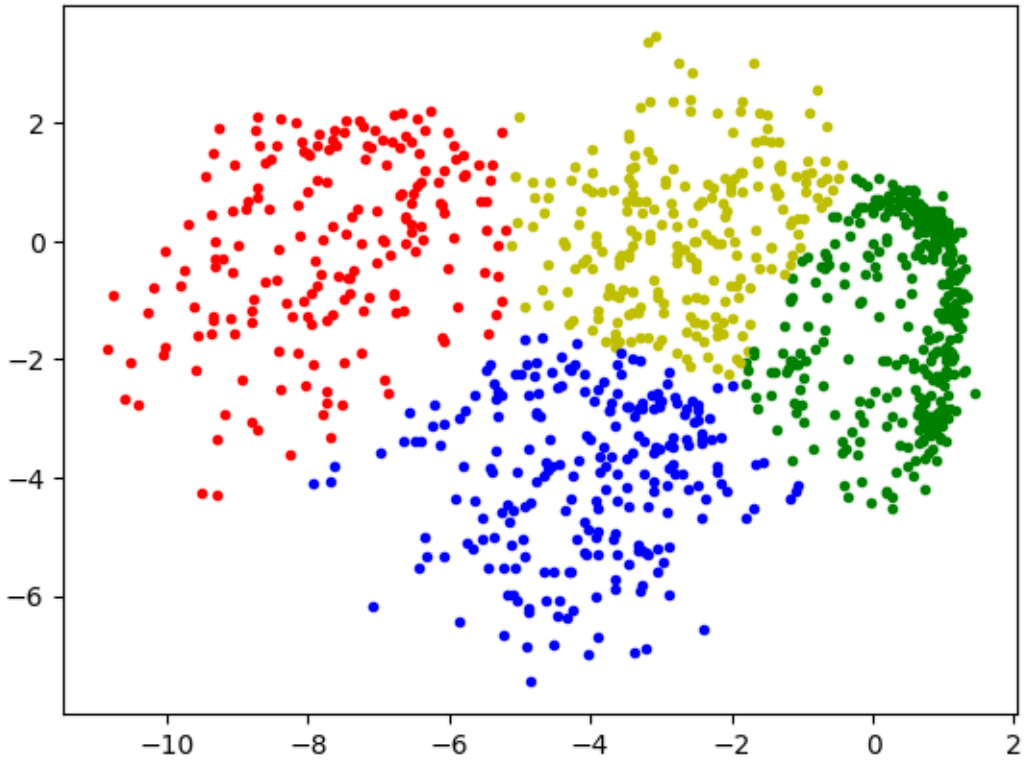
[ ]:
```
# do not change
def accuracy_score(y_true, y_pred):
    """
    Calculate clustering accuracy.
        y_true: true labels, numpy.array with shape `(n_samples,)`
        y_pred: predicted labels, numpy.array with shape `(n_samples,)`
    # Return
        accuracy, in [0,1]
    """
    y_true = np.squeeze(y_true)
    y_pred = np.squeeze(y_pred)
    assert y_true.shape == y_pred.shape, 'illegal inputs'

    y_true = y_true.astype(np.int64)
    assert y_pred.size == y_true.size
    D = max(y_pred.max(), y_true.max()) + 1
    w = np.zeros((D, D), dtype=np.int64)
    for i in range(y_pred.size):
        w[y_pred[i], y_true[i]] += 1
    from scipy.optimize import linear_sum_assignment
    row_ind, col_ind = linear_sum_assignment(w.max() - w)
    return sum([w[row_ind[i], col_ind[i]] for i in range(len(row_ind))]) * 1.0 /
 ↪ y_pred.size
```

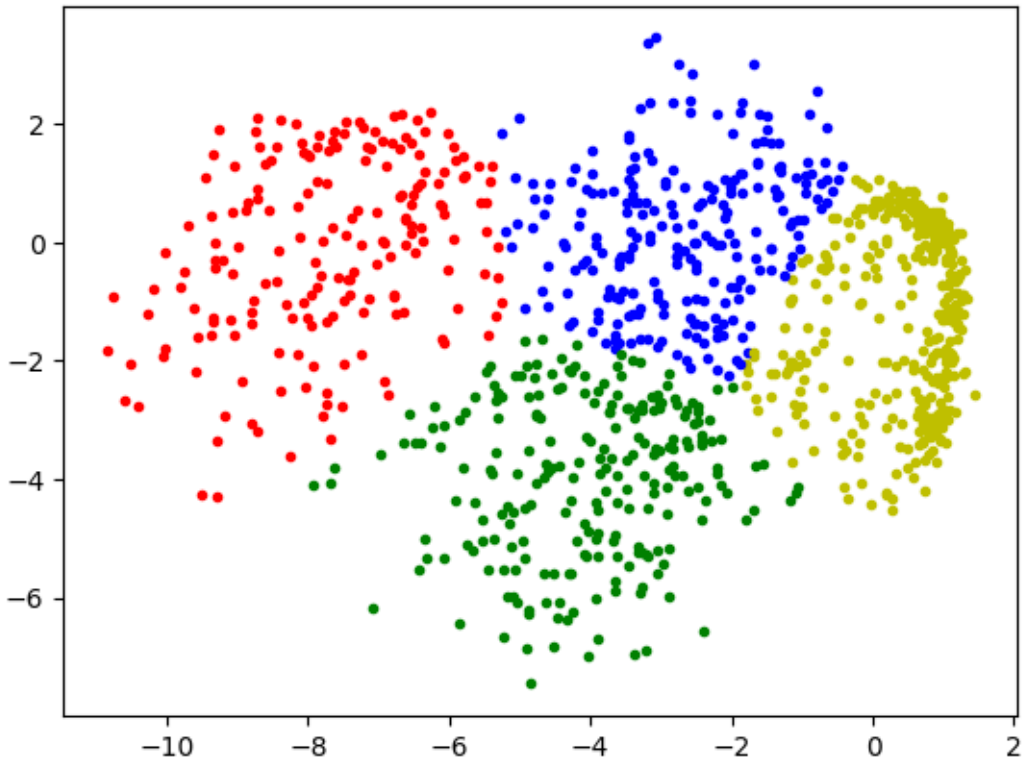
[ ]:
```
best_accuracy = 0
best_labels = None

# run kmeans 10 times with different random initializations
```

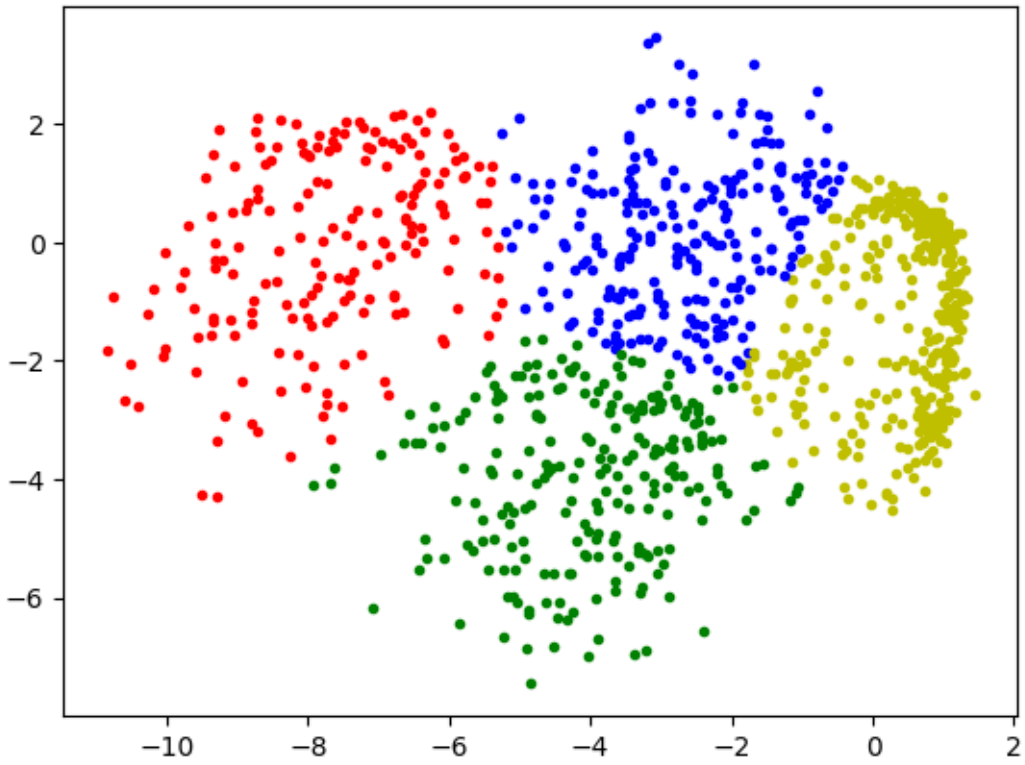
```python
for run in range(10):
    # run kmeans with current random state
    labels = kmeans(X2d, k=4, max_iter=500, random_state=run)

    # plot the clustering result
    plot_with_colors(X2d, labels)

    # calculate accuracy using accuracy_score
    accuracy = accuracy_score(y, labels)

    # track the best accuracy and labels
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_labels = labels

# output the best clustering accuracy achieved
print(f'Best Accuracy: {best_accuracy}')
```

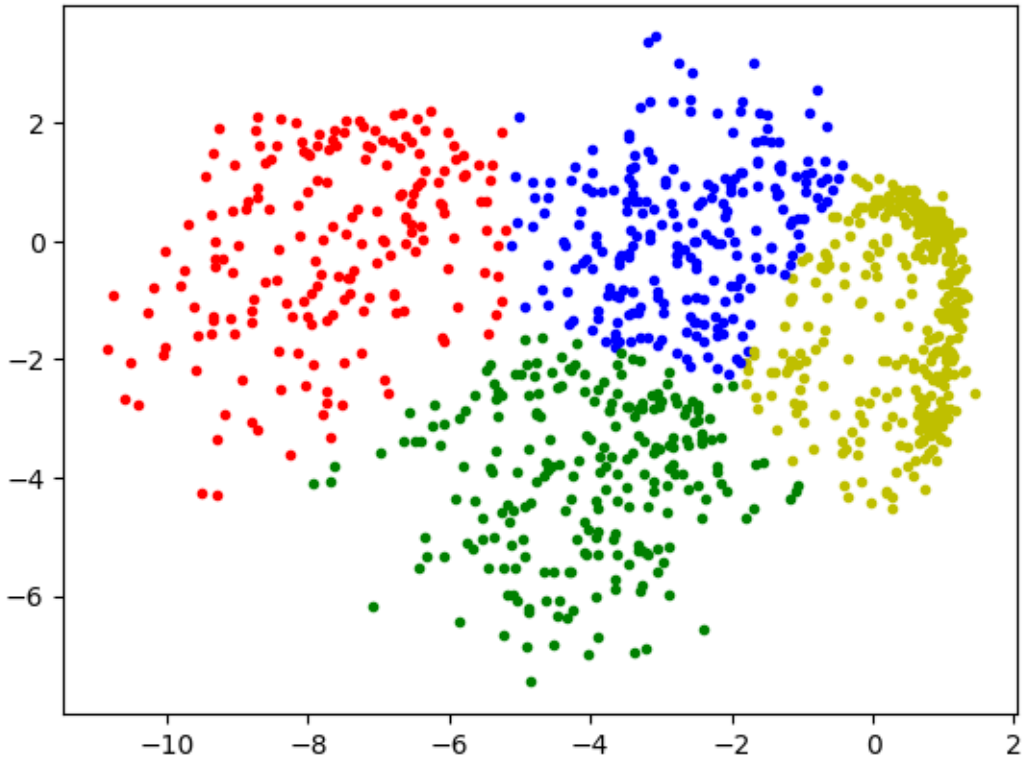


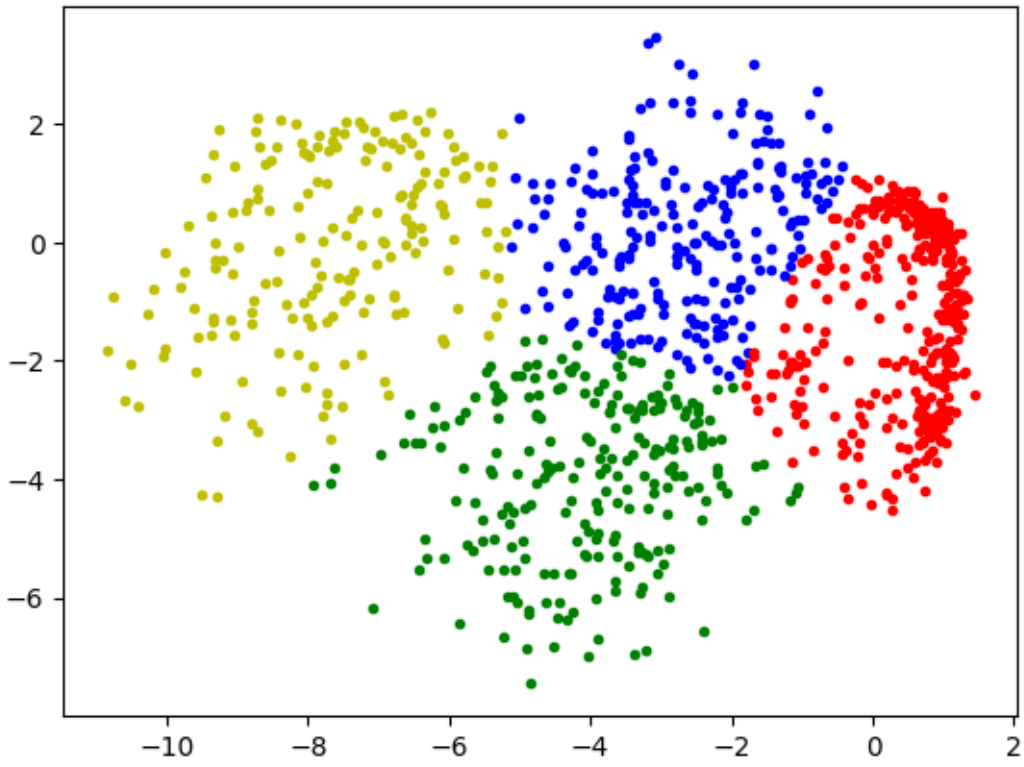

2
0
−2
−4
−6
−10
−8
−6
−4
−2
0
2

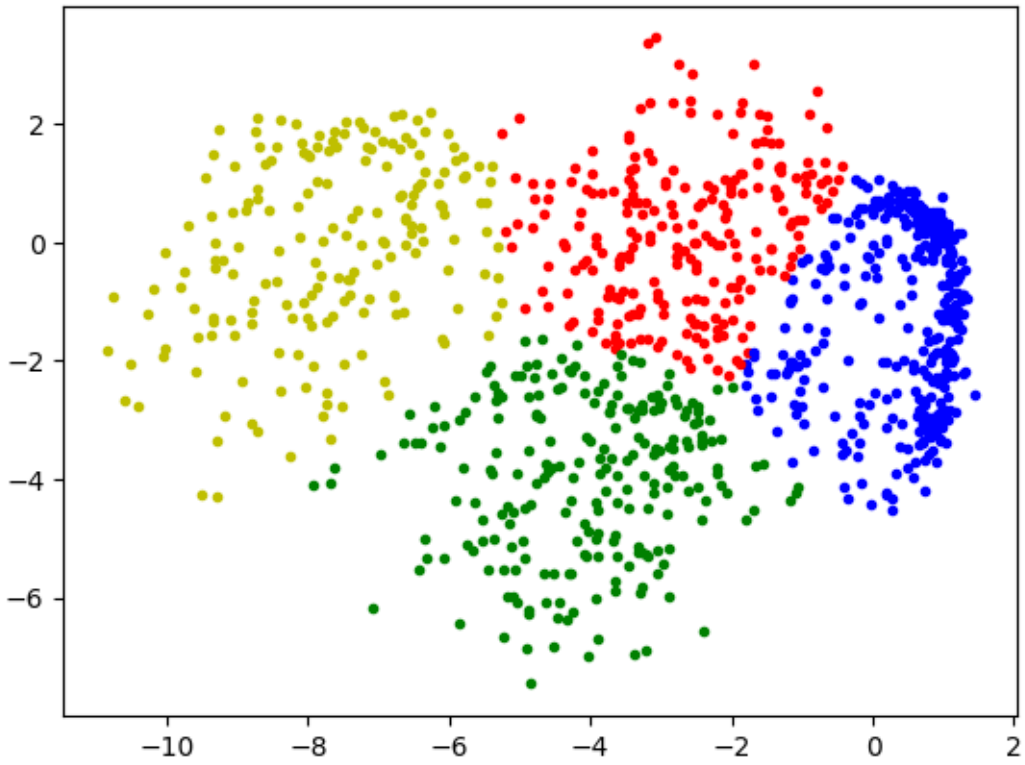

2
0
−2
−4
−6
−10
−8
−6
−4
−2
0
2

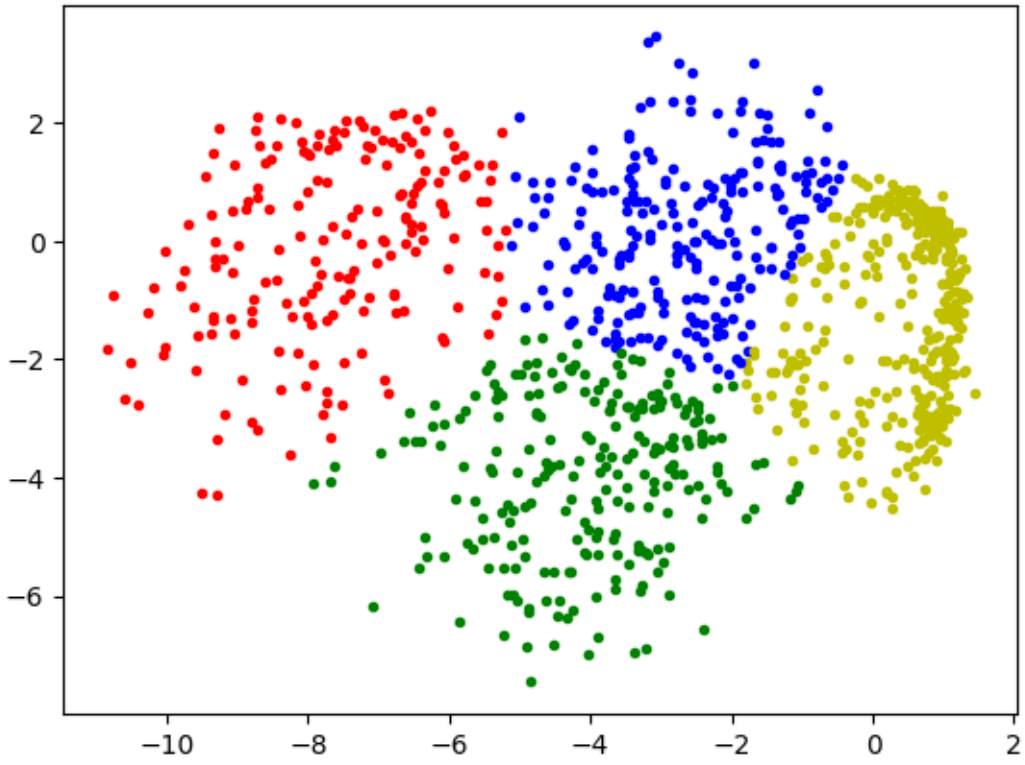

2
0
−2
−4
−6
−10
−8
−6
−4
−2
0
2

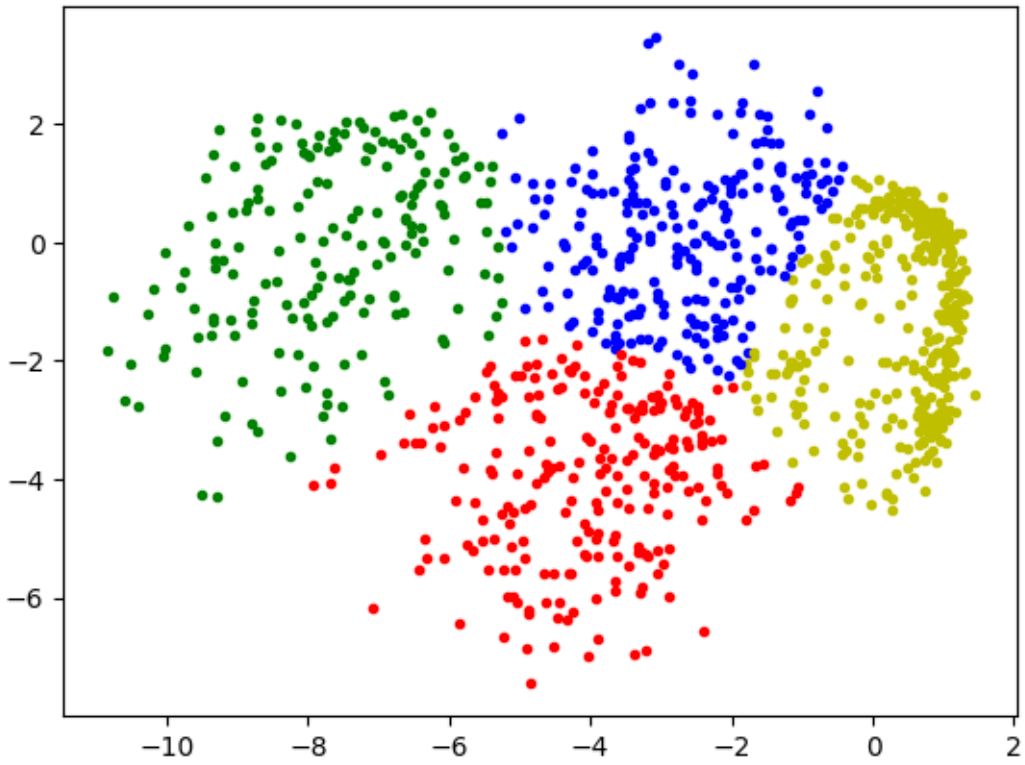

2
0
−2
−4
−6
−10
−8
−6
−4
−2
0
2


2
0
−2
−4
−6
−10
−8
−6
−4
−2
0
2


2
0
−2
−4
−6
−10
−8
−6
−4
−2
0
2


2
0
−2
−4
−6
−10
−8
−6
−4
−2
0
2


2
0
−2
−4
−6
−10
−8
−6
−4
−2
0
2




```
Best Accuracy: 0.745
```

We have been testing k-means on the top two principal components for the purpose of visualization. Please run k-means on the (784 dimensional) original image dataset (again using 4 clusters). Try at least 10 different random initializations and report the best accuracy as above.

[ ]:
```
best_accuracy = 0
best_labels = None

#run kmeans 10 times on the original dataset
for run in range(10):
    # run kmeans with current random state
    labels = kmeans(X, k=4, max_iter=500, random_state=run)

    # calculate clustering accuracy using accuracy_score
    accuracy = accuracy_score(y, labels)

    # track the best accuracy and labels
    if accuracy > best_accuracy:
        best_accuracy = accuracy
```

```
        best_labels = labels

# output the best clustering accuracy achieved
print(f'Best Accuracy on Original Dataset: {best_accuracy}')
```

```
Best Accuracy on Original Dataset: 0.861
```