Assignment 4

The goal of this assignment is to run some experiments with scikit-learn on a fairly sizeable and interesting image data set. This is the MNIST data set that consists of lots of images, each having 28x28 pixels. By today's standards, this may seem relatively tiny, but only a few years ago was quite challenging computationally, and it motivated the development of several ML algorithms and models that are now state-of-the-art solutions for much bigger data sets.

The assignment is experimental. We will try to whether a combination of PCA and kNN can yield any good results for the MNIST data set. Let's see if it can be made to work on this data set.

Note: There are less difficult Python parts in this assignment. You can get things done by just repeating things from the class notebooks. But your participation and interaction via Canvas is always appreciated!

Preparation Steps

```
# Import all necessary python packages
import numpy as np
#import os
#import pandas as pd
#import matplotlib.pyplot as plt
#from matplotlib.colors import ListedColormap
#from sklearn.linear_model import LogisticRegression
```

```
# we load the data set directly from scikit learn
#
# note: this operation may take a few seconds. If for any reason it fails we
# can revert back to loading from local storage.

from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split


X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
y = y.astype(int)
X = ((X / 255.) - .5) * 2
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=10000, random_state=123, stratify=y)

```

Question 1. Inspecting the Dataset

(i) How many data points are in the training and test sets ?

(ii) How many attributes does the data set have ?

Exlain how you found the answer to the first two questions.

[**Hint**: Use the 'shape' method associated with numpy arrays.]

(iii) How many different labels does this data set have. Can you demonsrate how to read that number from the vector of labels *y_train*?

(iv) How does the number of attributes relates to the size of the images?

(v) What is the role of line 12 in the above code?

(Please insert cells below for your answers. Clearly id the part of the question you answer)

```
 1 # (i)
 2 print("(i)")
 3 print("Data points in training set =", X_train.shape[0])
 4 print("Data points in test set =", X_test.shape[0])
 5 # (ii)
 6 print("(ii)")
 7 print("Attributes in data set =", X_train.shape[1])
 8 # (iii)
 9 print("(iii)")
10 print("Number of different labels =", len(np.unique(y)))
11 # (iv)
12 print("(iv)")
13 print("28 pixel * 28 pixel greyscale image =", X_train.shape[1], "attributes total.")
```

```
(i)
Data points in training set = 60000
Data points in test set = 10000
(ii)
Attributes in data set = 784
(iii)
Number of different labels = 10
(iv)
28 pixel * 28 pixel greyscale image = 784 attributes total.
```

(v)

Line 12 normalizes the pixel (feature) values to be between -1 and 1. This is a linear transformation.

```
1
```

```
1 # For grader use only
2
3 G = [0]*5
4
5
```

```
# insert grade here  (out of 1o - 2 points for each question)
# G[1] =


```

Question 2. PCA on MNIST

Because the number of attributes of the MNIST data set may be too big to apply kNN on it (due to the 'curse of dimensionality'), we want to compress the images down to a smaller number of 'fake' attributes.

Use scikit-learn to output a data set *X_train_transformed* and *X_test_transformed*, with l attributes. Here a reasonable choice of l is 10, equal to the number of labels. But you can try slightly smaller or bigger values as well.

Hint: Take a look at lecture-4 class notebook, and imitate what we did there. Be careful though, to use only the scikit-learn demonstration, not the exhaustive PCA steps we did before it.

Note: This computation can take a while. If problems are encountered we can try the same experiment on a downsized data set.

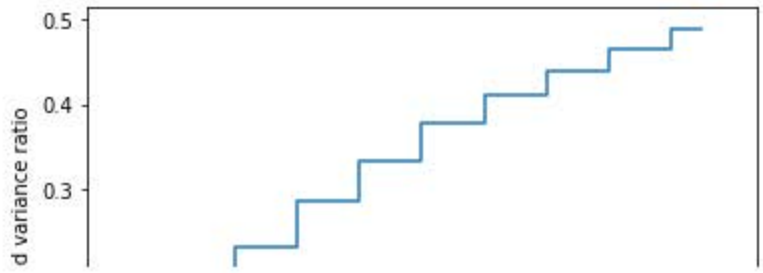
```
from sklearn.decomposition import PCA
```

```
n_components = 10
pca = PCA(n_components = n_components)
X_train_transformed = pca.fit_transform(X_train)
pca.explained_variance_ratio_
```

```
array([0.09764215, 0.07164792, 0.06143184, 0.0539996 , 0.04891746,
       0.04308324, 0.03274778, 0.02881822, 0.0275126 , 0.02348185])
```

```
import matplotlib.pyplot as plt
```

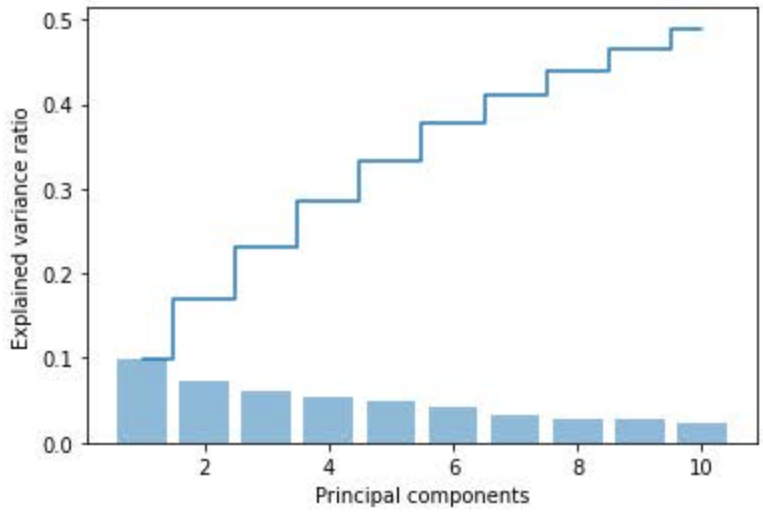
```
plt.bar(range(1, n_components + 1), pca.explained_variance_ratio_, alpha=0.5, align='cente
plt.step(range(1, n_components + 1), np.cumsum(pca.explained_variance_ratio_), where='mid'
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal components')
plt.show()
```




```
pca = PCA(n_components = n_components)
X_test_transformed = pca.fit_transform(X_test)
pca.explained_variance_ratio_
```

```
array([0.09646816, 0.07110542, 0.06196344, 0.05438917, 0.04876163,
       0.04292087, 0.03305321, 0.02945667, 0.02792599, 0.02324967])
```

```
plt.bar(range(1, n_components + 1), pca.explained_variance_ratio_, alpha=0.5, align='cente
plt.step(range(1, n_components + 1), np.cumsum(pca.explained_variance_ratio_), where='mid'
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal components')
plt.show()
```




```
print(X_train_transformed.shape)
print(X_test_transformed.shape)
```

```
(60000, 10)
(10000, 10)
```

Double-click (or enter) to edit

```
# for grader use

# insert grade here (out of 4)
# G[2] =
```

Question 3. kNN on 'fake'-attributes MNIST

Having calculated the *transformed* MNIST data set we can now apply a kNN approach to the MNIST classification data set. Here are the sets:

(i) Fit a k-NN classifier on the transformed data set. Here k is a hyperparameter, and you can experiment with it. Be aware though, that larger k can take more time to fit.

(ii) Apply the classifier on the transformed test set. What is the classification accuracy?

(iii) A theoretical question: if we skipped all the above steps and we just assigned a **random** label to each test point, what would the classification accuracy be on average? Does your result (ii) beat the random expectation?

(iv) Experiment with different settings of k, and other hyperparameters that are described in the scikit-learn manual of the kNN classifier. Report your findings in a separate cell. Also for **participation points**: report your best result on Canvas!

[**Hint**: Imitate the steps from the classroom notebook]

Note to Grader: An important detail here is that the PCA transformation obtained from the test data, must be the same one that is applied to the test data. So, running fit_transform separately on training and test will not give the best results, simply because the second application forgets the training data and only works with the test data. This is the reason I am using the 'copy' input argument in PCA. This is a potential mistake than can be replicated on all experiments, so maybe you can subtract a fixed small penalty from each. Similarly, we should be careful that kkn uses the correct inputs for training and testing.

```
from sklearn.neighbors import KNeighborsClassifier

# (i) and (ii)
pca = PCA(n_components = 10, copy = False)
X_train_transformed = pca.fit_transform(X_train)
X_test_transformed = pca.transform(X_test)
knn = KNeighborsClassifier(n_neighbors = 2, p = 5, metric = 'minkowski')
knn.fit(X_train_transformed, y_train)
knn.score(X_test_transformed, y_test)
```

```
0.8294
```

(iii) The expected accuracy with random labels is 10%.

(iv) Note to grader: The above is an indicative solution with a specific choice of parameters. In general, it is expected that the students did a series of similar experiments. You can also look and give extra points for creative experiments like the one below that found that simple knn with $k = 1$ gives a very good accuracy

```
knn = KNeighborsClassifier(n_neighbors = 1, p = 2, metric = 'minkowski')
knn.fit(X_train, y_train)
knn.score(X_test, y_test)
```

```
0.9733
```

```
1
```

```
# for grader use

# insert grade here (out of 12)
# G[3] =
```

```
# for grader use

# Total Grade Calculation
# T = sum(G)
```