Lab1 (5IR - 2024): Human Activity Recognition (HAR) - Supervised learning

Group name: 2

Names: Jumin, Gauché

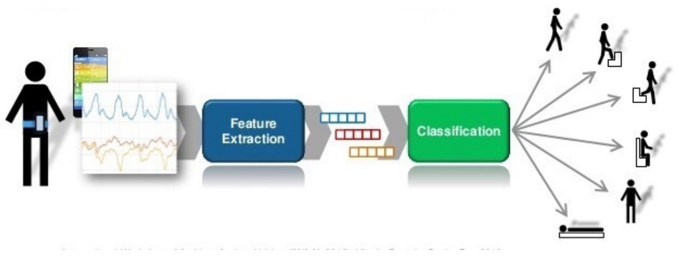
First names: Noël, Clément

In this project, we will try to predict 8 different human activities:

1-Recumbent, 2-Sitting, 3-Standing, 4-Walking, 5-Biking, 6-Nordic Walking, 7-Cleaning_Aspirator or 8-Repassing

using different sensors (accelerometer, gyroscope, magnetometer or temperature) placed at different points on the body (hand, torso and ankle).

For the accelerometer, gyroscope and magnetometer sensors, measurements in ms^{-2}, rad/s and μT respectively are taken on the 3-axes at a sampling rate of 100Hz. Each measurement is labeled.



For each sensor and position, a matrix of the measurements collected is provided, along with the activity label (i.e. the ground truth).

After loading the data, this lab is divided into 3 parts:

- Part I: Implementing DTW to create a k-NN recognition system;
- Part II: Dimension reduction using PCA and classification using neural networks;
- Part III: (Bonus) Dimension reduction using k-medoids for DTW + k-NN.

[!!!!!] What you have to do is indicated as **Questions** and **#Todo.** [!!!!!]

In [1]:
```
# Necessary libraries: ipykernel matplotlib==3.7.3 pandas scikit-learn seabo

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import math
from scipy import stats

from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.decomposition import PCA
from sklearn.neural_network import MLPClassifier

import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
```

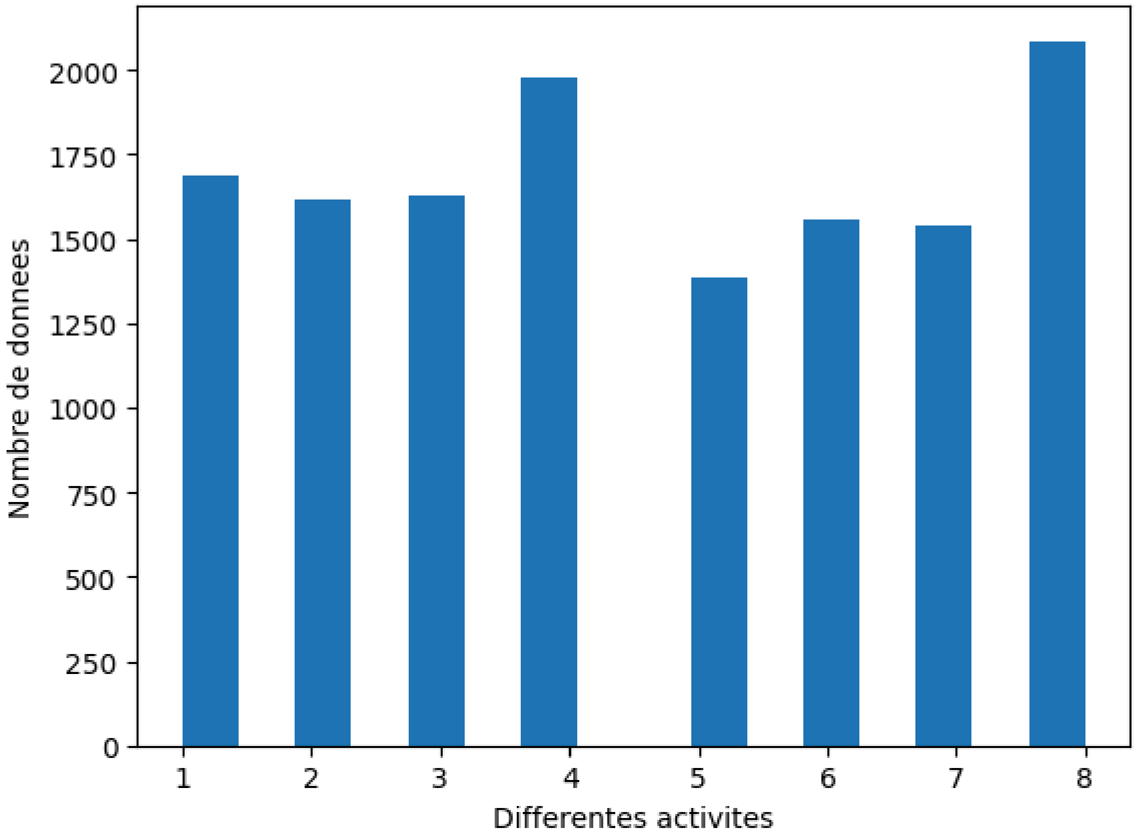
Preprocessing data

Read the data files for all sensors and the labels

In [2]:
```
# Data loading
datas = []

# Sensors part for Temperature
datas.append(np.swapaxes(np.load('DataTemperature.npy'), 1, 2))

# Sensors part (Accelerometer, Gyroscope and Magnetometer) placed on the Har
# Hand
datas.append(np.load('DataHandAcc.npz')['DataHandAcc'])
datas.append(np.load('DataHandGyro.npz')['DataHandGyro'])
datas.append(np.load('DataHandMagneto.npz')['DataHandMagneto'])
#TODO: do the same for the other sensors: accelerator/gyrometer/magnetometer
datas.append(np.load('DataAnkleAcc.npz')['DataAnkleAcc'])
datas.append(np.load('DataAnkleGyro.npz')['DataAnkleGyro'])
datas.append(np.load('DataAnkleMagneto.npz')['DataAnkleMagneto'])
datas.append(np.load('DataChestAcc.npz')['DataChestAcc'])
datas.append(np.load('DataChestGyro.npz')['DataChestGyro'])
datas.append(np.load('DataChestMagneto.npz')['DataChestMagneto'])
```

In [3]:
```
RawData=np.concatenate(datas, axis=2)
```

In [4]:
```
# Loading the ground truth: labels
fl=np.load('labels.npz')
labels=fl['labels']

# Repartition of data by class
plt.hist(labels,bins=16)
plt.xlabel('Differentes activites')
plt.ylabel('Nombre de donnees')
plt.show()
```



Scaling the data and smoothing the temporal series

In [5]:
```
N, T, S = RawData.shape
X_reshaped = RawData.reshape(N, T*S)

scaler = StandardScaler()
X_scaled_2d = scaler.fit_transform(X_reshaped)
Data = X_scaled_2d.reshape(N, T, S)
```

In [6]:
```
# Smoothing function
def smooth_signal(X, k):
    for i in range(len(X)):
        Xi = X[i]
        downsample=math.ceil(len(Xi)/k)
        for j in range(downsample):
            Xj = Xi[j*k:min((j+1)*k, len(Xi)),:]
            X[i,j] = np.mean(Xj, axis=0)
    X = X[:,:downsample,:]
    return X

# Smooth training and testing data
SMOOTH_RATIO=100
X = smooth_signal(Data, SMOOTH_RATIO)
```

Questions (look inside the data)

-How many examples in the dataset ?

In [7]:
```
print(f'Number of elements in the dataset: {N}')
```

```
Number of elements in the dataset: 13477
```

-What is the difference between the variables RawData, Data, and X ?

- RawData= RawData from the files.

- Data= RawData transform in a normalized version to be understand by the libraries.
- X= A smoothed version of Data to help the model to be less long to calculate.

About Data (or RawData):
-Considering the sensors give values with a frequency of 100Hz, what is the length of the measurement window for each example ?

In [8]:
```
frequency = 100
print(f'The measurement window for a frequency of {frequency}Hz is {T/freque
```
```
The measurement window for a frequency of 100Hz is 5.0s
```

-For each example, for each time step, what is the dimension of the data i.e. how many sensors ?
There is 3 IMU:

- One on the wrist for the dominant arm.
- One on the chest.
- One on the ankle for the dominant side.

Each IMU has 4 modules:

- Temperature (°C).
- Acceleration data in 3D (ms^{-2}) => One sensor on x,y and z.
- Gyroscopic data in 3D (rad/s) => One sensor on x,y and z.
- Magnetometer data in 3D (μT) => One sensor on x,y and z. So there is (3+3+3+1)*3=30 sensors.

In [9]:
```
print(f'There is {S} sensors with {N} examples for each of them')
```
```
There is 30 sensors with 13477 examples for each of them
```

About X: what is the new frequency of the measurement ?

In [10]:
```
print(f'The new frequency of measurement is {frequency/SMOOTH_RATIO}Hz')
```
```
The new frequency of measurement is 1.0Hz
```

Part I-a: Implementing the dynamic programming algorithm

1. Implement the Dynamic Time Warping algorithm (DTW) in a python function such that:

- the local distance used is a parameter which is a callable function with 2 inputs and 1 output: e.g. euclidian distance(a, b)
- local distances can be computed with any type of input local data: e.g. (a,b)=('A','T'), (a,b)=(12,22), (a,b)=((1,2,3), (4,5,6))
- there are two optionnal parameters: local and global constraints

The DTW algorithm is given and explained in the Lecture 3... Slides 12-16.

In [11]:
```
# Define distances functions
def euclidian_distance(a, b):
    return np.linalg.norm(a - b)

def dist_letter(a, b):
    #print(a,b,a==b)
    return 1 if a!=b else 0

# Define the Dynamic Time Warping algorithm
def dtw(a, b, distance, w, gamma):
    N= len(a)
    M= len(b)
    g = np.full((N + 1, M + 1), np.inf)
    g[0, 0] = 0
    for i in range(1, N+1):
        for j in range(1, M+1):
            if abs(i-j) > gamma:
                continue
            local_d = distance(a[i-1], b[j-1])
            i_1 = g[i-1,j]+(w[0]*local_d)
            i_2 = g[i-1,j-1]+(w[1]*local_d)
            i_3 = g[i,j-1]+(w[2]*local_d)
            g[i,j] = min(i_1, i_2, i_3)
    return (g[N,M]/(N+M))
```

1. Test your methods on the examples given in class. Typically:

- example 1:
  - DTW between (4, 6, 1) and (5, 1) => resulting score S=0.4.
  - local distance: the euclidian distance
  - local constraints: (1,1,1)
  - global constraint: 2
- example 2:
  - DTW between "ATGGTACGTC" and "AAGTAGGC" => resulting score S=1/6
  - local distance: 1 if the characters are different, 0 else
  - local constraints: (1,1,1)
  - global constraint: 4

In [12]:
```
# Example 1:
a = (4, 6, 1)
b = (5, 1)

print(dtw(a,b,euclidian_distance,(1,1,1),2))

# Example 2:
a = ('A', 'T', 'G', 'G', 'T', 'A', 'C', 'G', 'T', 'C')
b = ('A', 'A', 'G', 'T', 'A', 'G', 'G', 'C')

print(dtw(a,b,dist_letter,(1,1,1),4))
```

```
0.4
0.16666666666666666
```

Question

- What are the effects of the local and global constraints on the DTW computation ?

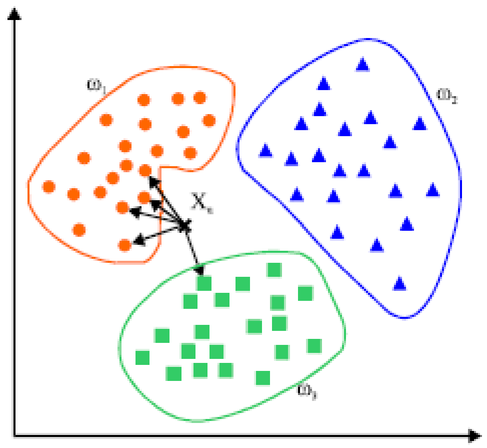
1. Local constraint(ω0, ω1, ω2): to add specific weighting on the different paths around the diagonal.

2. Global constraints(gamma): we limit the path to a certain band around the diagonal. The higher the constraints are, the faster the calculation will be because it will avoid computing the other values that are too far from the diagonal.

Part I-b: DTW human activity recognition system

In this section, we want to use the k-Nearest Neighbor method (k-NN) combined with the DTW algorithm in order to construct a model for the classification of human acivity based on the previously loaded data.

The principle of our approach is the following: 1) For each class, compute the DTW score between the test temporal profile and the temporal profiles of the class. Test with data from only one sensor then include them all if possible. 2) Determine the class membership by k-nearest neighbors.



We provide here the class KNN overlaying the k-NN algorithm implemented inside the scikit-learn library (*sklearn.neighbors.KNeighborsClassifier*):

- In scikit-learn, only one dimensional data is accepted for classification,
- Here, we use multiple sensors, thus two dimensional data for each example which can be treated easily using the **DTW algorithm with the euclidian distance**.

In scikit-learn, KNeighborsClassifier takes as argument *metrics* which can be a callable function taking as input 2 arrays representing 1D vectors and returning one value indicating the distance between those vectors. **We provide a metric function using the DTW function you implemented which should be called *dtw(...)*.**

In [13]:
```
def dtw_KNN(x1, x2):
    n=x1.shape[0]
    folds=Data.shape[2]
    x1 = x1.reshape(int(n/folds),folds)

    n=x2.shape[0]
    folds=Data.shape[2]
    x2 = x2.reshape(int(n/folds),folds)

    res=dtw(x2, x1, w=(1,1,1), gamma=5, distance=euclidian_distance)
    return res

class KNN:
    def __init__(self, n_neighbors=1, metric=dtw_KNN, n_jobs=-1):
        self.n_neighbors=n_neighbors
        self.metric=metric
        self.n_jobs=n_jobs
        self.clf=n_jobs
        self.clf = KNeighborsClassifier(n_neighbors, metric=self.metric, n_j

    def fit(self, xtrain, y_train):
        m, n, k = xtrain.shape
        xtrain = xtrain.reshape(m,n*k)
        self.clf.fit(xtrain, y_train)
        return self.clf

    def predict(self, xtest):
        m, n, k = xtest.shape
        xtest = xtest.reshape(m,n*k)
        return self.clf.predict(xtest)

    def score(self, xtest, ytest):
        m, n, k = xtest.shape
        xtest = xtest.reshape(m,n*k)
        return self.clf.score(xtest, ytest)
```

Information: Here are the main commands:

- The line *clf = KNN(n_neighbors=1])* creates an object of type classifier based on the n_neighbors closest neighbors,
- The instruction *clf.fit(X, y)* uses the data to define the classifier (training),
- The command *clf.predict()* is used to classify the new examples,
- The command *clf.predict_proba()* allows to estimate the probability of the proposed classification,
- The command *clf.score(xtest, ytest)* computes the global score of the classifier for a given dataset.

Training and test datasets

Randomly create the training and test datasets:

- The training dataset must be of size 160. Be careful to select examples such that the labels are balanced (20 examples for each activity)
- The test set of size 1000.

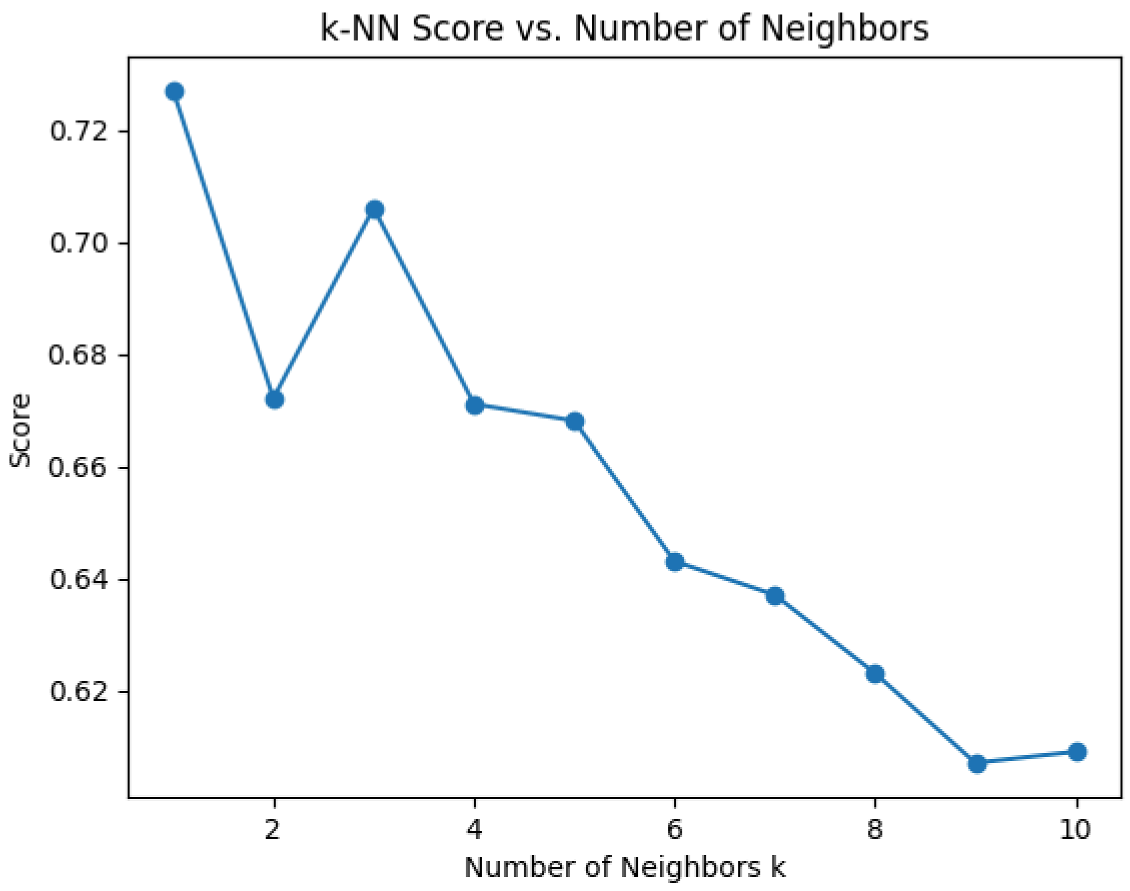
In [14]:
```
np.random.seed(42)
# Get the index where each labels start
indices = [np.where(labels == i)[0] for i in range (1,8)]
# We get 20 random values for training
train_indices = np.hstack([np.random.choice(class_indices, 20, replace=False
# We get 1000 other data for test
test_indices = np.random.choice(np.setdiff1d(np.arange(len(labels)), train_i

x_train, x_test = X[train_indices], X[test_indices]
y_train, y_test = labels[train_indices], labels[test_indices]
```

Create and train the k-NN model

Try several choices for the number of neighbors k (e.g. 1 to 10). and select the one giving best results on the test set. Ideally, plot the evolution of the score on the test set depending on k

In [15]:
```python
# Create and train the k-NN model
k_values = range(1, 11)
scores = []

for k in k_values:
    clf = KNN(n_neighbors=k)
    clf.fit(x_train, y_train)
    score = clf.score(x_test, y_test)
    scores.append(score)
    print(f'k={k}, Score={score}')

# Plot the evolution of the score on the test set depending on k
plt.plot(k_values, scores, marker='o')
plt.xlabel('Number of Neighbors k')
plt.ylabel('Score')
plt.title('k-NN Score vs. Number of Neighbors')
plt.show()
```

```
k=1, Score=0.727
k=2, Score=0.672
k=3, Score=0.706
k=4, Score=0.671
k=5, Score=0.668
k=6, Score=0.643
k=7, Score=0.637
k=8, Score=0.623
k=9, Score=0.607
k=10, Score=0.609
```



Evaluation

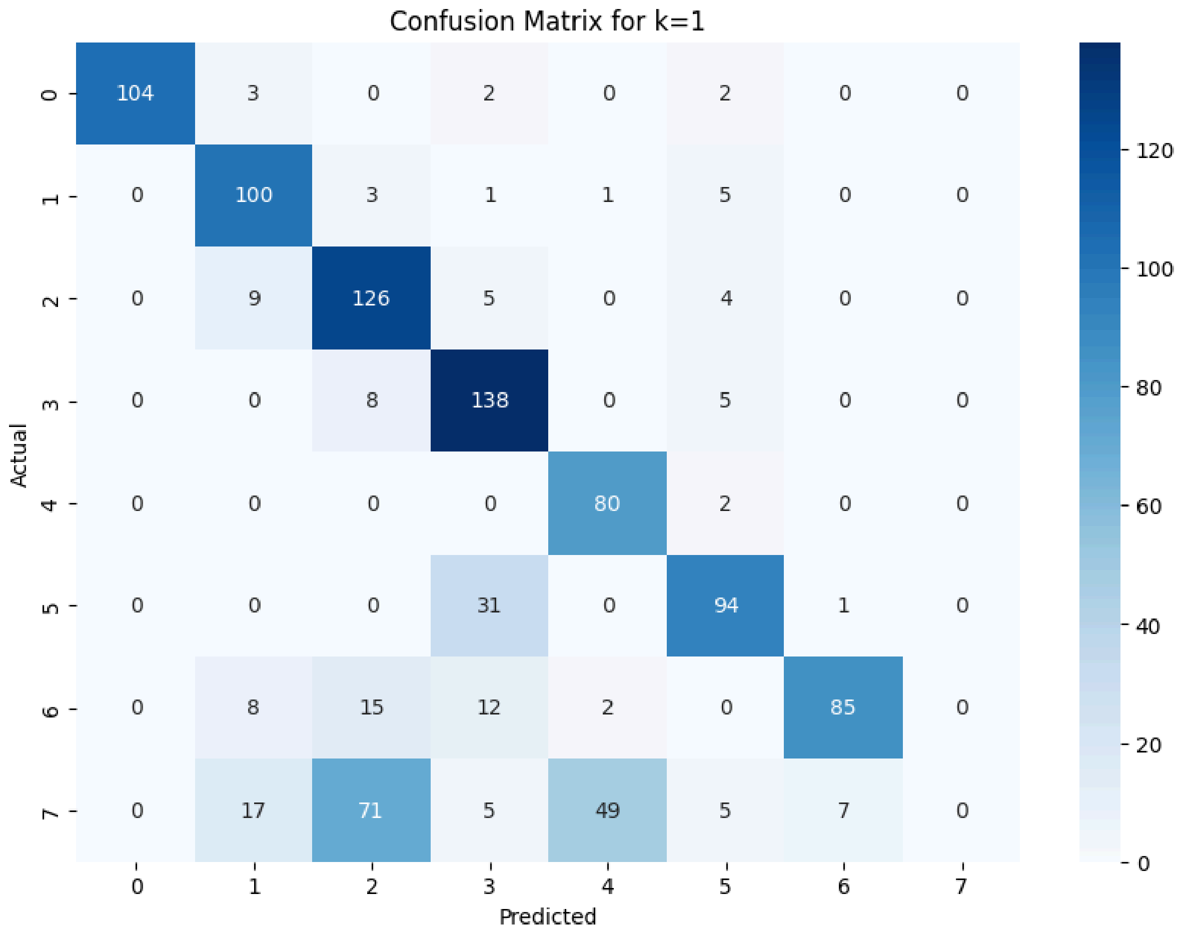
From the prediction on the test set obtained after DTW:

- compute the confusion matrix that counts the number of right and wrong classifications.
- compute the score on the training and test sets.

In [16]:
```python
# Test for a k=1
k=1
# Compute the confusion matrix that counts the number of right and wrong cla
clf = KNN(n_neighbors=k)
clf.fit(x_train, y_train)
score = clf.score(x_test, y_test)
scores.append(score)
y_pred = clf.predict(x_test)
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.title(f'Confusion Matrix for k={k}')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

# Compute the score on the training and test sets.
print(f'Training score: {clf.score(x_train, y_train)}')
print(f'Test score: {clf.score(x_test, y_test)}')

# Test for a k=10
k=10
# Compute the confusion matrix that counts the number of right and wrong cla
clf = KNN(n_neighbors=k)
clf.fit(x_train, y_train)
```

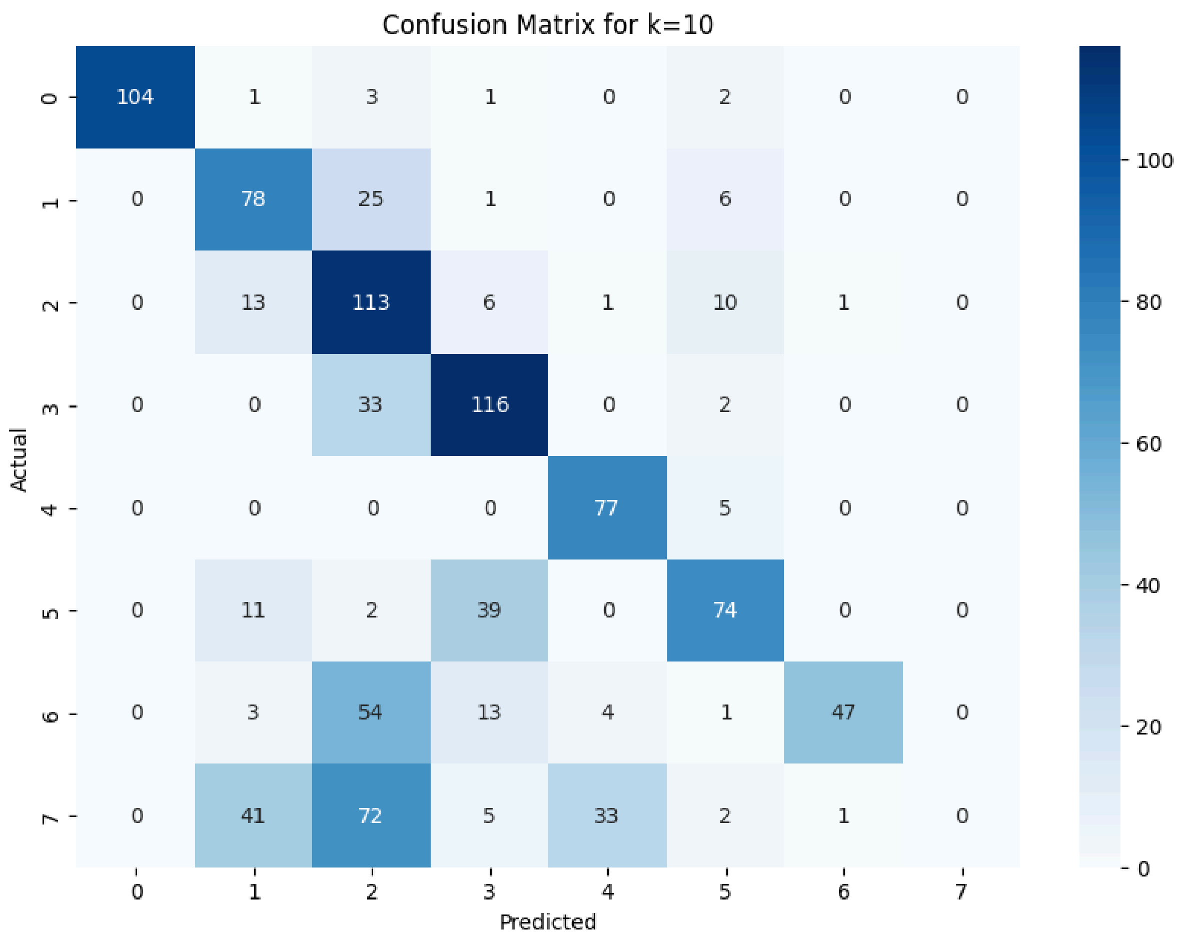
```
score = clf.score(x_test, y_test)
scores.append(score)
y_pred = clf.predict(x_test)
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.title(f'Confusion Matrix for k={k}')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

# Compute the score on the training and test sets.
print(f'Training score: {clf.score(x_train, y_train)}')
print(f'Test score: {clf.score(x_test, y_test)}')
```

```
Confusion Matrix
```



Actual \ Predicted	0	1	2	3	4	5	6	7
0	104	3	0	2	0	2	0	0
1	0	100	3	1	1	5	0	0
2	0	9	126	5	0	4	0	0
3	0	0	8	138	0	5	0	0
4	0	0	0	0	80	2	0	0
5	0	0	0	31	0	94	1	0
6	0	8	15	12	2	0	85	0
7	0	17	71	5	49	5	7	0

```
Training score: 1.0
Test score: 0.727
Confusion Matrix
```



Actual \ Predicted	0	1	2	3	4	5	6	7
0	104	1	3	1	0	2	0	0
1	0	78	25	1	0	6	0	0
2	0	13	113	6	1	10	1	0
3	0	0	33	116	0	2	0	0
4	0	0	0	0	77	5	0	0
5	0	11	2	39	0	74	0	0
6	0	3	54	13	4	1	47	0
7	0	41	72	5	33	2	1	0

```
Training score: 0.8
Test score: 0.609
```

Questions:

- When creating the *sklearn.neighbors.KNeighborsClassifier* model in KNN(), what is the purpose of the n_jobs=-1 parameter ?

  The description from scikit learn is perfect:

  ```
    n_jobsint, default=None
        The number of parallel jobs to run for neighbors search.
  None means 1 unless in a joblib.parallel_backend context. -1
  means using all processors. See Glossary for more details.
  Doesn’t affect fit method.
  ```
  So, n_jobs=-1 is simply to specify to the model to use every processors or thread available for the calculation part.

- Usually, ~80% of the data is used as train set, why do we have to choose a very small number of examples for this purpose here ?

  The nearest neighbors method does not actually require a training dataset like the other models. These data are only used during the prediction phase, which limits the number of data needed. Additionally, the k-NN method requires calculating the distance between each data and all other data to determine its nearest neighbors, which involves computing n^2 distances, which is too computationally intensive.

- What is the best choice for number of neighbors ?

As you can see, we got a best score with k=1 than with k=10. It's due to the fact that the k-NN method will only search one neighboor so it will be extremely precise.

- Analyse the results (accuracy, execution time, ...)

  The execution time on the k-NN method is really long because it need to compute n^2 distances.
  Concerning the accuracy, as we can see on the matrixs, for class 7 and 8, the results are really bad. Especially with class 8 that is not even recognized as a class 8. This can come from noise on the dataset or maybe the fact that the value captured for class 8 are really near to the class 3 and 5.

- Briefly, what are the advantages and disadvantages of the k-NN method: optimality ? computation time ? scalability ?

  - Advantages:
    - Simplicity: Easy to implement and understand.
    - No training phase: Unlinke machine learning models like MLP, k-NN doesn't require real training, everything is done during the test phase.
    - Adaptability to local patterns: k-NN can capture complex patterns in the data, even with non-linear boundaries like handwriting for example.
    - No sensitivity to outliers during Training: Since there is no really training phase, there is no problem from the training phase.
  - Disadvantages:
    - Can be expensive: need to compute (n^2) distances and store all data points.
    - Sensitive to non-pertinent and correlated features: If the dataset contains non-pertinent or correlated features, this can distort the distances measures, reducing accuracy.
    - Sensitive to outliers during Testing: Testing is the only phase so it can be sensitive to testing phase problem like outliers data.
    - Parameter: Distance is long to calculate and it needs to implement a version for each type of dataset like we have done with letters.

Part II: Comparison of dynamic programming with a neural network classification method after PCA dimension reduction

In this section, we will compare the results of the classification using k-NN and DTW with those of another classification method: dimension reduction using PCA combined with neural networks (Multi-Layer Perceptron).

We'll be using the functions for calculating PCA and MLPClassifier via the *scikit-learn* python library.

Training and test datasets

Starting from the data in the variable X:

- For each example, transform the data to be 2D, i.e. we want to concatenate the measurement of all sensors (one can use numpy.reshape).
- Randomly create the training and test datasets with 80% of the data in the training dataset (there is a function for this..).

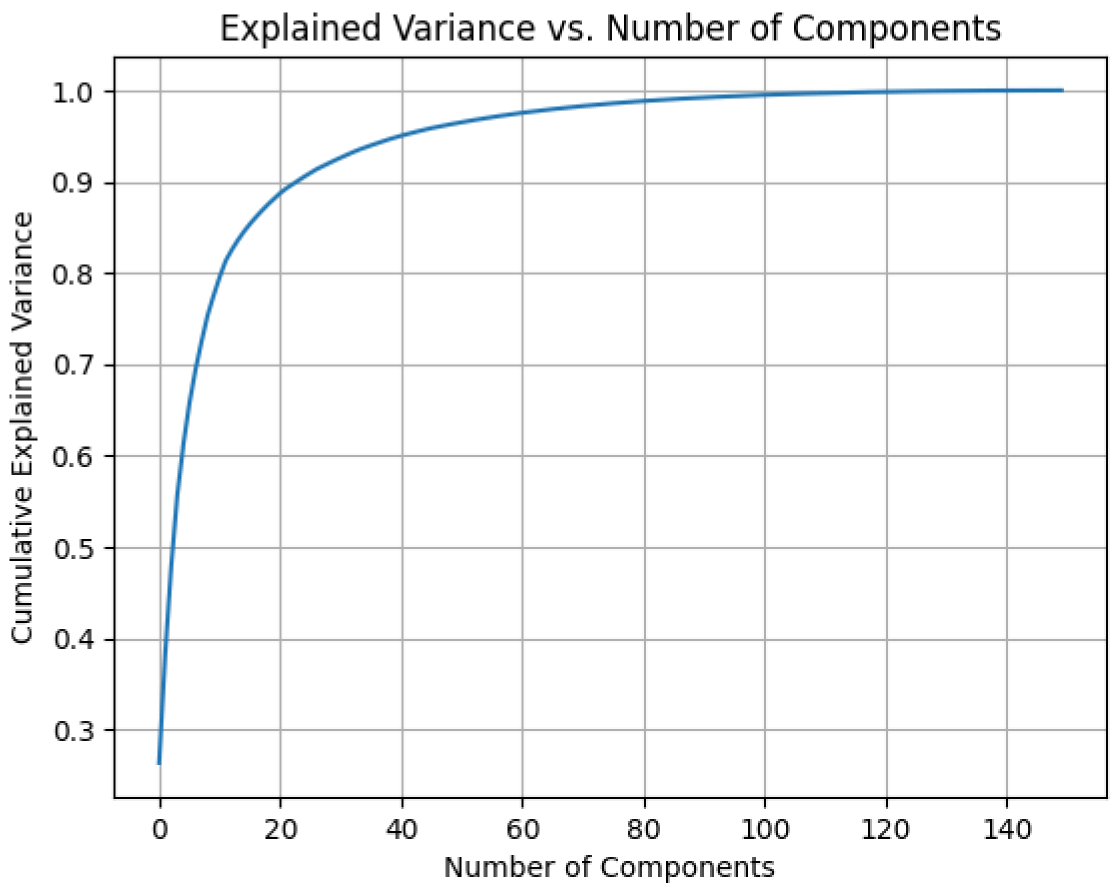
In [17]:
```
# For each example, transform the data to be 2D, i.e. we want to concatenate
x_reshaped = X.reshape(X.shape[0], -1)
# Randomly create the training and test datasets with 80% of the data in the
x_train, x_test, y_train, y_test = train_test_split(x_reshaped, labels, test
```

Determining the best number of components for PCA

We want to determine the ideal number of components for the dimension reduction, i.e. a reduced number of variables reprensenting well the data, using using the *PCA* function in the *scikit-learn* library:

- *pca=PCA()*: Create a PCA model
- *pca.fit(x)*: From the training data, calculate the p principal axes of the PCA by extracting the p eigenvectors, denoted X_1, X_2,..., X_p, associated with the p largest eigenvalues of the variance-covariance matrix variance-covariance matrix Σ_{App}. These eigenvectors will form the new database.
- Get the explained variance ratio of each principal component (see the documentation) and plot these
- Determine how many principal components must be kept to explain 99% of the variance.

In [18]:
```
pca = PCA()
pca.fit(x_train)

# Get the explained variance ratio of each principal component
explained_variance_ratio = pca.explained_variance_ratio_

plt.plot(np.cumsum(explained_variance_ratio))
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Explained Variance vs. Number of Components')
plt.grid()
plt.show()

n_components = np.argmax(np.cumsum(explained_variance_ratio) >= 0.99) + 1
print(f'It needs {n_components} components to explain 99% of the variance')
```



```
It needs 85 components to explain 99% of the variance
```

PCA pre-processing

Using the previously determined number of components:

- *pca=PCA(n_components=[...])*: Create the PCA model with right number of components
- *pca.fit(x)*: the model must be trained on the training data again
- *pca.transform(x)*: Project the data from the training and test database into this new database by multiplying each vector vector by the base $P = [X_1 X_2 ... X_p]$.

In [19]:
```
pca = PCA(n_components=n_components)
pca.fit(x_train)

x_train_pca = pca.transform(x_train)
x_test_pca = pca.transform(x_test)
```

Questions

- What is the number of components kept ?

In [20]:
```
print(f'We passed, using this PCA, from {x_test.shape[1]} components to {x_t
```

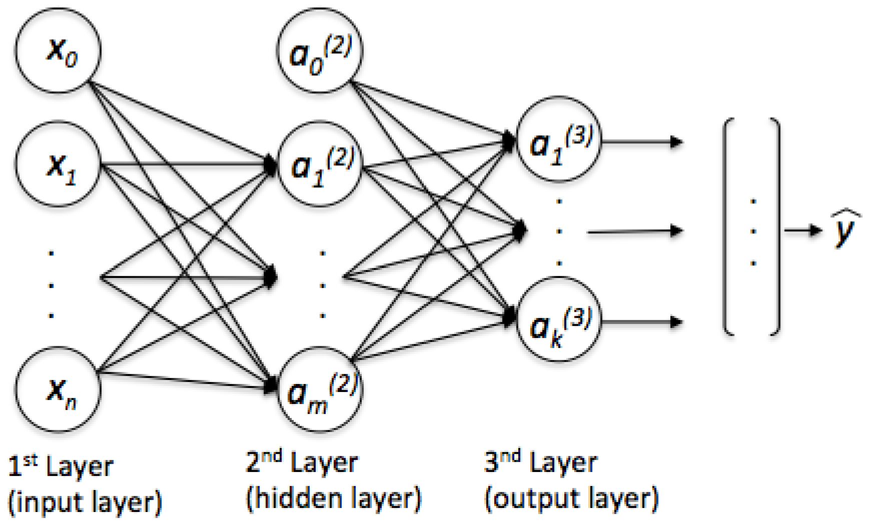
```
We passed, using this PCA, from 150 components to 85 components
```

- Consider this: the data for each example is a 1D vector where each value (feature) is the measurement of 1 sensor at 1 time step. In your opinion, does it make sense to use this as features and apply dimension reduction with PCA directly? Do you expect good results for the upcoming classification?

Using PCA on a 1D vector where each value represents a measurement from a sensor may not always be the most appropriate approach. However, it can be beneficial if the data are correlated, as it remove redundancy. Fortunately, in our case, the data are correlated, which enables us to enjoy the advantages of PCA by reducing the number of features to process and eliminate noise. Therefore, we can expect better results compared to the k-NN method, as k-NN is prone to errors when the data is too correlated or too noisy.

Classification using neural networks

We will use the library **sklearn.model_selection** to develop ANN of type Multi Layer Perceptron (MLP). We use the class: MLPClassifier.



This model optimizes the cross entropy function (loss function) and a gradient based method. The main parameters of this class are:

- **hidden_layer_sizes** is a tuple that specifies the number of neurons in each hidden layer; from the entrance to the exit. For example, a unique hidden layer of 55 neurons, hidden_layer_sizes = (55); for three hidden layers of size respectively 50, 12 and 100 neurons, hidden_layer_sizes = (50, 12, 100).
- **activation** defines the activation function for hidden layers: {"identity", "logistic", "tanh", "relu"}, default "relu"
  - ‘identity’, no-op activation, linear bottleneck, returns f(x) = x
  - ‘logistic’, logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
  - ‘tanh’, hyperbolic tan function, returns f(x) = tanh(x).
  - ‘relu’, rectified linear unit function, returns f(x) = max(0, x)
- **max iter**, default =200, indicates the max number of iterations of the solver.

The main commands are:

- *mlp = MLPClassifier(hidden_layer_sizes=(10,5,), max_iter=300, activation='relu')*
- *mlp.fit(xtrain, ytrain)*

- *ypred = mlp.predict(xtest)*

Accuracy of the model can eventually be computed using the library **sklearn.metrics** and the commands:

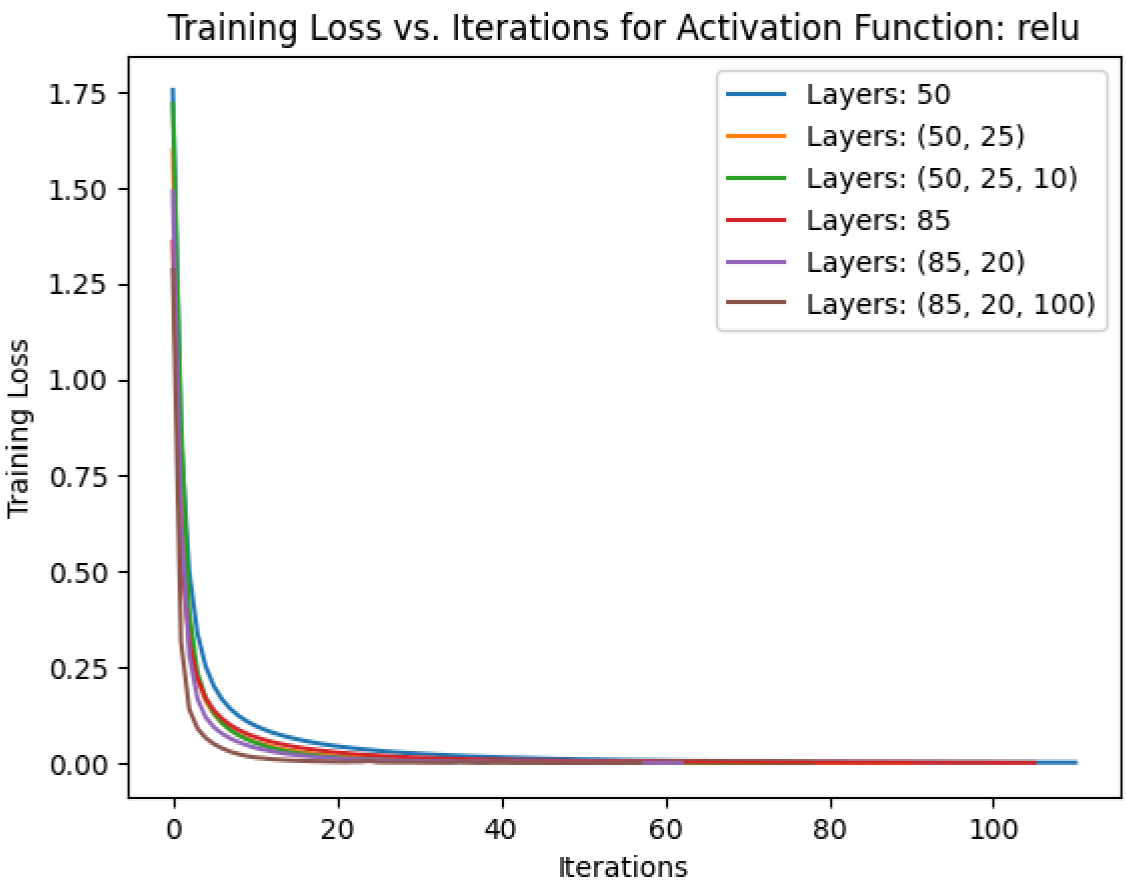
- Score: accuracy_score(y_test, y_pred)
- Report: classification_report(y_test, y_pred))

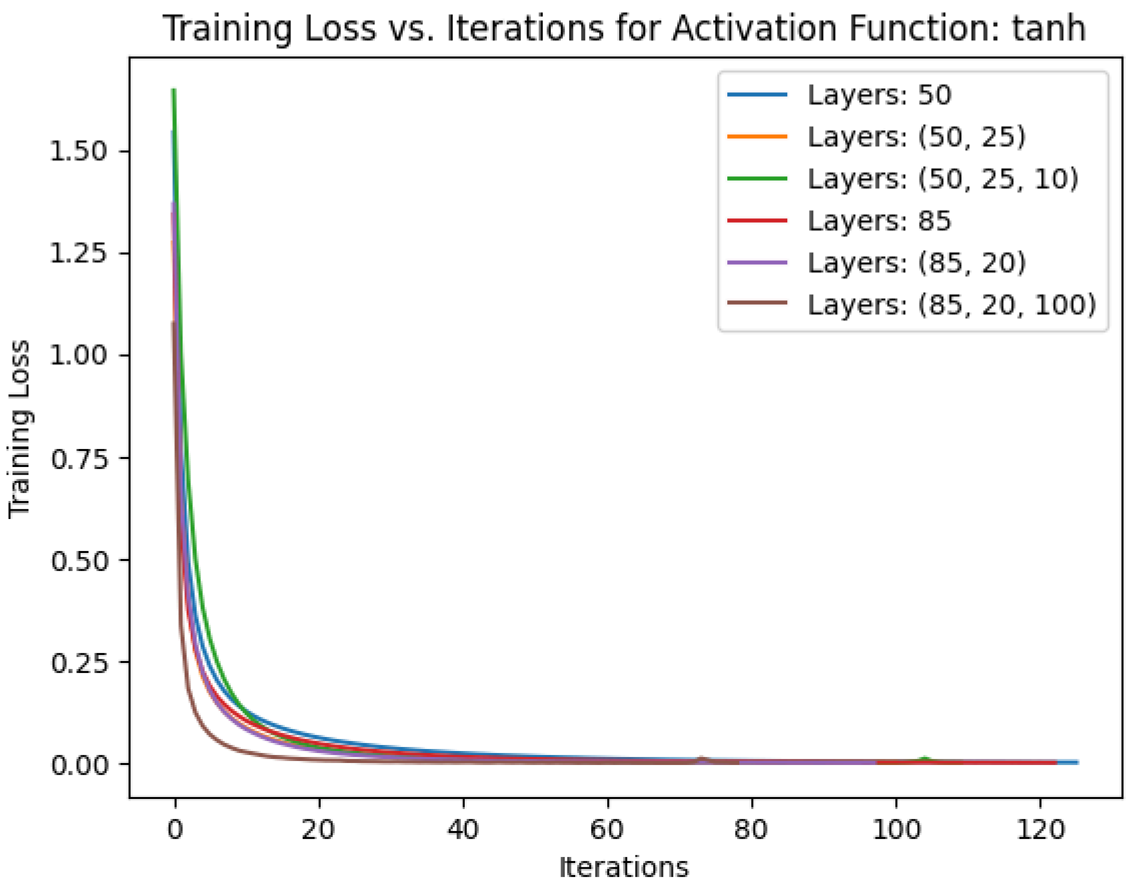
The usual sklearn learning and testing functions are as follows: (**fit**, **predict**, **score**).

Create and train the MLP model

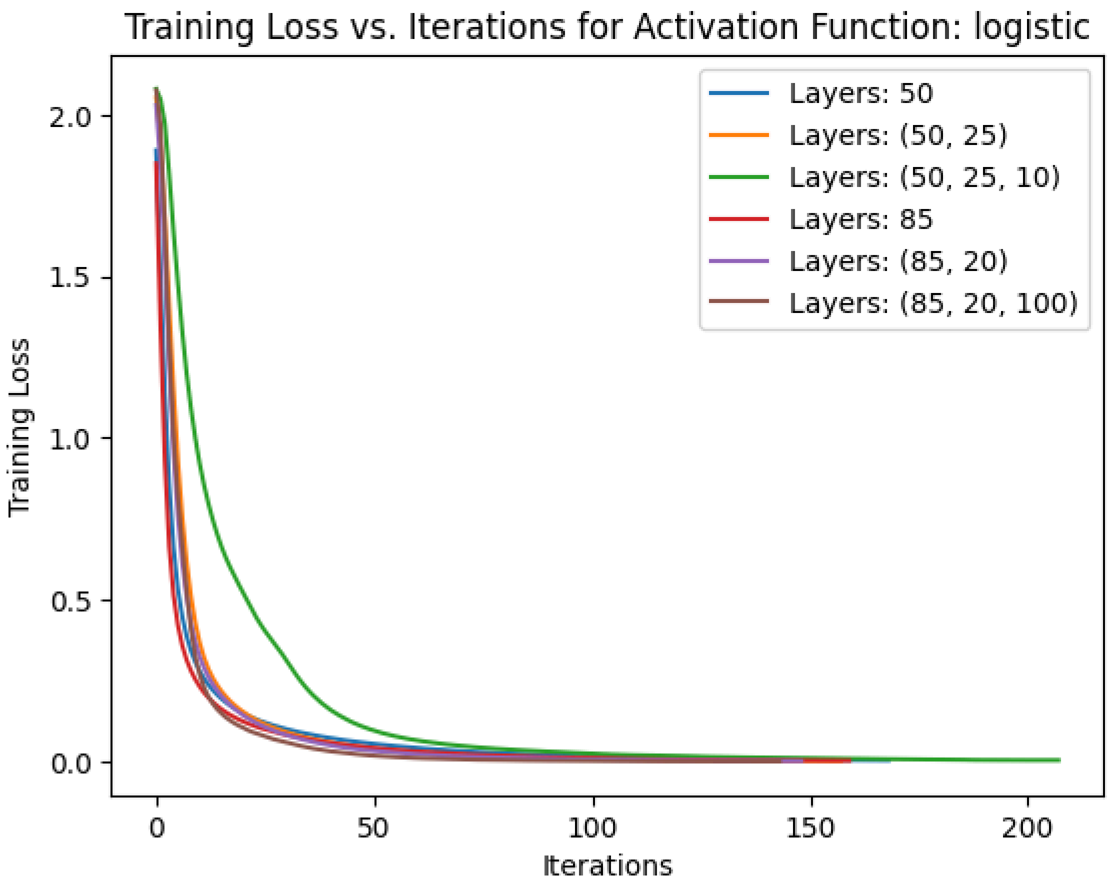
- Try several choices for the number of layers and the activation function.
- Plot the evolution of the training score during the training (can be obtained from the model, see the documentation).

In [21]:
```
import time
# Define different configurations for the MLP
layer_configs = [(50), (50, 25), (50, 25, 10), (85), (85, 20), (85, 20, 100)
activation_functions = ['relu', 'tanh', 'logistic', 'identity']

best_score = 0
best_config = None
best_mlp = None
results = []

for activation in activation_functions:
    plt.figure()
    for layers in layer_configs:
        start_time = time.time()
        mlp = MLPClassifier(hidden_layer_sizes=layers, max_iter=300, activat
        mlp.fit(x_train_pca, y_train)
        end_time = time.time()

        # Plot the evolution of the training score
        plt.plot(mlp.loss_curve_, label=f'Layers: {layers}')

        # Evaluate the model
        y_pred_mlp = mlp.predict(x_test_pca)
        score = accuracy_score(y_test, y_pred_mlp)

        # More data to determine overfitting/underfitting
        y_pred_mlp_train = mlp.predict(x_train_pca)
        train_score = accuracy_score(y_train, y_pred_mlp_train)

        if score > best_score:
            best_score = score
            best_config = (layers, activation)
            best_mlp = mlp
            best_y_pred_mlp = y_pred_mlp

        results.append({
            'Layers': layers,
            'Activation Function': activation,
            'Computation Time (s)': end_time - start_time,
            'Test accuracy': score,
            'Train accuracy': train_score
        })
    plt.xlabel('Iterations')
    plt.ylabel('Training Loss')
    plt.title(f'Training Loss vs. Iterations for Activation Function: {activ
    plt.legend()
    plt.show()

# Create a DataFrame to display the results
results_df = pd.DataFrame(results)
print(results_df)

print(f'Best MLP Configuration is with layers: {best_config[0]}, and the {be
print(f'Best Score: {best_score}')
```

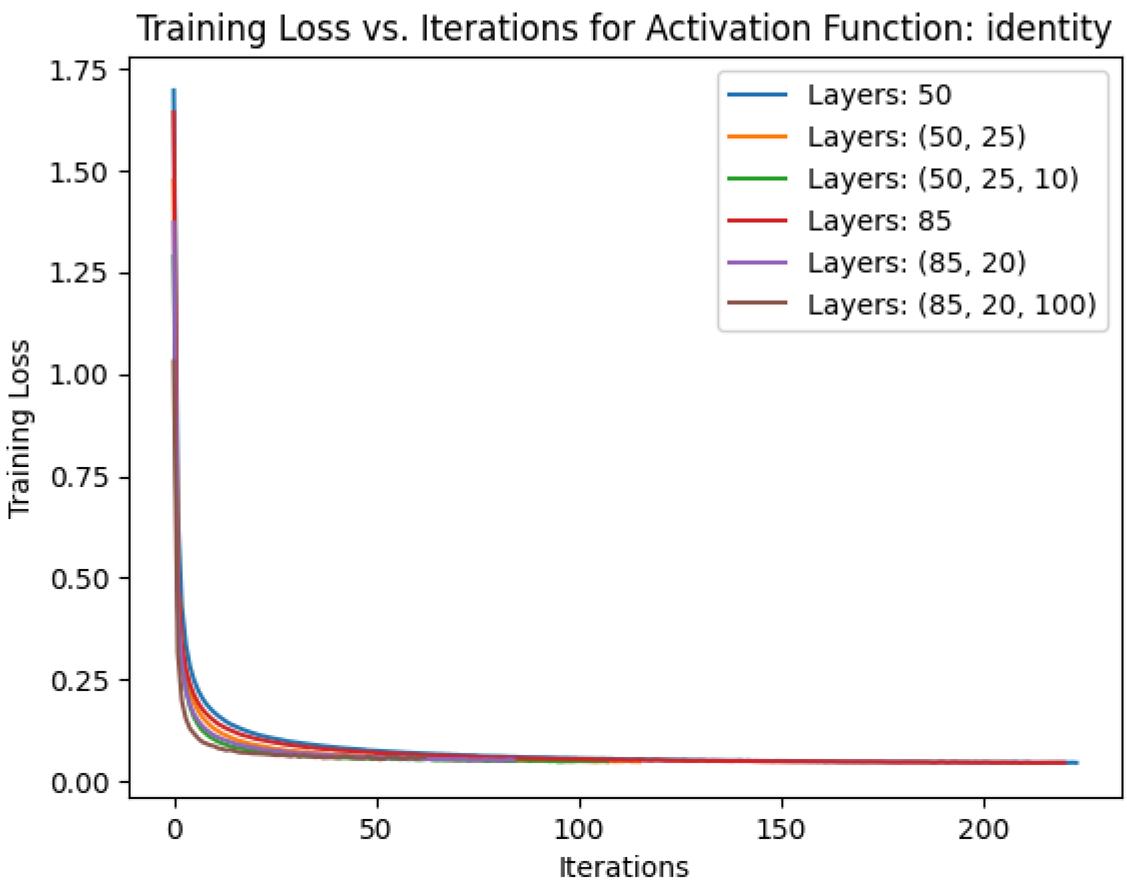




Training Loss vs. Iterations for Activation Function: tanh
Training Loss
Iterations
0.00
0.25
0.50
0.75
1.00
1.25
1.50
0
20
40
60
80
100
120
Layers: 50
Layers: (50, 25)
Layers: (50, 25, 10)
Layers: 85
Layers: (85, 20)
Layers: (85, 20, 100)



Training Loss vs. Iterations for Activation Function: logistic
Training Loss
Iterations
0.0
0.5
1.0
1.5
2.0
0
50
100
150
200
Layers: 50
Layers: (50, 25)
Layers: (50, 25, 10)
Layers: 85
Layers: (85, 20)
Layers: (85, 20, 100)


Training Loss vs. Iterations for Activation Function: identity
Layers: 50
Layers: (50, 25)
Layers: (50, 25, 10)
Layers: 85
Layers: (85, 20)
Layers: (85, 20, 100)
Training Loss
Iterations
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
0
50
100
150
200

```
           Layers Activation Function  Computation Time (s)  Test accuracy  \
0              50                relu              3.901959       0.982567   
1        (50, 25)                relu              6.247088       0.983309   
2    (50, 25, 10)                relu              5.199544       0.982567   
3              85                relu              4.662297       0.987760   
4        (85, 20)                relu              4.638242       0.987018   
5   (85, 20, 100)                relu              5.351333       0.981083   
6              50                tanh              4.134591       0.985163   
7        (50, 25)                tanh              4.419186       0.985163   
8    (50, 25, 10)                tanh              5.666066       0.984792   
9              85                tanh              4.616278       0.982567   
10       (85, 20)                tanh              4.952225       0.985534   
11  (85, 20, 100)                tanh              5.363963       0.985534   
12             50            logistic              6.885749       0.980712   
13       (50, 25)            logistic              7.458453       0.981454   
14   (50, 25, 10)            logistic             11.461787       0.977374   
15             85            logistic              9.140348       0.982938   
16       (85, 20)            logistic             10.703159       0.984421   
17  (85, 20, 100)            logistic             14.961763       0.982938   
18             50            identity              7.775272       0.964021   
19       (50, 25)            identity              4.789637       0.965134   
20   (50, 25, 10)            identity              6.809754       0.966617   
21             85            identity              9.718411       0.965134   
22       (85, 20)            identity              3.946857       0.959570   
23  (85, 20, 100)            identity              4.206963       0.963279   

    Train accuracy  
0         0.999814  
1         1.000000  
2         1.000000  
3         0.999814  
4         0.999907  
5         0.999722  
6         0.999814  
7         0.999814  
8         0.999907  
9         0.999907  
10        1.000000  
11        1.000000  
12        0.999536  
13        0.999814  
14        0.999629  
15        0.999814  
16        0.999907  
17        1.000000  
18        0.990075  
19        0.990446  
20        0.988962  
21        0.990075  
22        0.987756  
23        0.986643  
Best MLP Configuration is with layers: 85, and the relu activation function
Best Score: 0.987759643916914
```

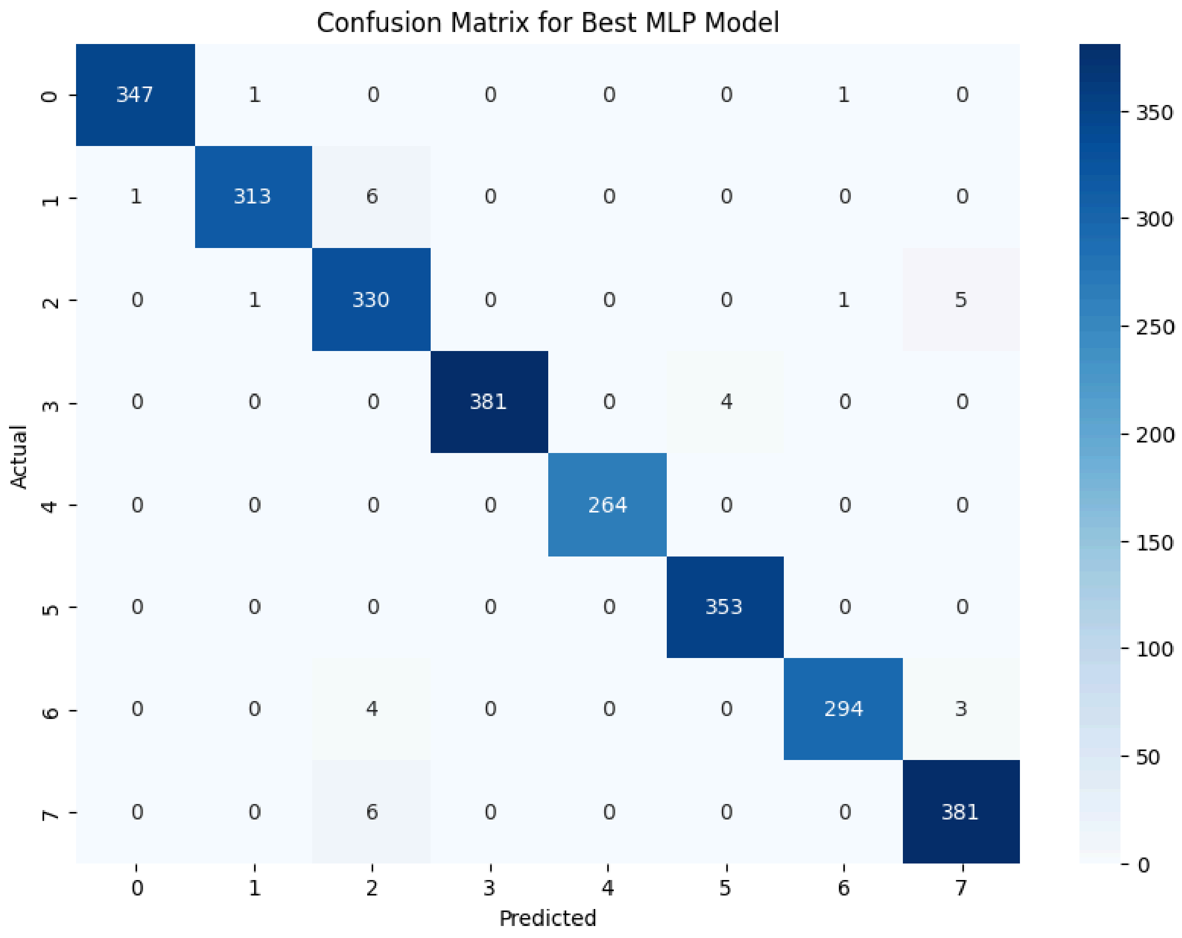
Evaluation

From the prediction on the test set obtained after DTW:

- compute the confusion matrix that counts the number of right and wrong classifications.
- compute the score on the training and test sets.

In [22]:
```
# Compute the confusion matrix that counts the number of right and wrong cla
print('Matrice de confusion par MLP')
conf_matrix_mlp = confusion_matrix(y_test, best_y_pred_mlp)
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix_mlp, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix for Best MLP Model')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

# Compute the score on the training and test sets.
print(f'Best train score: {best_mlp.score(x_train_pca, y_train)}')
print(f'Best test score: {best_mlp.score(x_test_pca, y_test)}')
```

```
Matrice de confusion par MLP
```



Actual \ Predicted	0	1	2	3	4	5	6	7
0	347	1	0	0	0	0	1	0
1	1	313	6	0	0	0	0	0
2	0	1	330	0	0	0	1	5
3	0	0	0	381	0	4	0	0
4	0	0	0	0	264	0	0	0
5	0	0	0	0	0	353	0	0
6	0	0	4	0	0	0	294	3
7	0	0	6	0	0	0	0	381

```
Best train score: 0.9998144884519061
Best test score: 0.987759643916914
```

Questions:

- What is the best model of MLP you found ? What did you try ?

  All the activation functions offered by the MLPClassifier function with different layers. Obviously my tests are not optimized, but this allowed me to obtain several pieces of information.

In [23]:
```
print(f'The best score is {best_score},and was obtained with the {best_confi
```

```
The best score is 0.987759643916914,and was obtained with the relu activati
on function and theses layers : 85
```

- Analyse the results (accuracy, execution time, Underfitting/Overfitting, ...)

Les résultats obtenus sont assez proches même si la fonction Relu est pour notre cas un excellent choix. Comme on peut le voir, de toutes les fonctions d'activation c'est celle qui est la plus régulière que ce soit dans le temps d'execution ou bien l'accuracy. De plus, les modèles obtenus ont tous obtenu un résultat très proche du score d'entrainement, prouvant de leur qualité. Comme on peut le voir pour le modèle choisi, celui-ci a une matrice de confusion proche de la perfection puisque très peu de donné ont été mal classé. Si l'on compare cette matrice à celle obtenu avec la méthode k-NN, où pour certaines classes, le résultat était catastrophique en plus d'être très long pour délivrer un résultat. La combinaison PCA+MLP est donc bien plus efficace pour notre dataset qu'avec la méthode des plus proches voisins.

- Briefly, what are the advantages and disadvantages of the MLP compared to k-NN method: optimality ? computation time ? scalability ?

  k-NN method is better in:

  - Simplicity: Compared to MLP which is quite abstract, k-NN is based on simple distances calculation which is more simplier to understand and implement.
  - No training phase, only test phase.

    MLP method is better in:

  - Computation time: A model creation is really fast compared to k-NN method.
  - Scalability: A model creation is really fast using MLP with PCA compared to k-NN method. Moreover, k-NN method can be slower if you add data because it needs to compute n^2 distances which can be really long if you got thousand and thousand of data.
  - Accuracy: In our case, the PCA method could be used and it helped a lot in reducing noise, and therefore, helped improve the accuracy of our model which is much higher than that obtained with the k-NN method.