SVM_NeuralNetworks_MNIST_dataset

December 21, 2023

Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE" and remove every line containing the expression: "raise …" (if you leave such a line your code will not run).

Do not remove any cell from the notebook you downloaded. You can add any number of cells (and remove them if not more necessary).

0.1 IMPORTANT: make sure to rerun all the code from the beginning to obtain the results for the final version of your notebook, since this is the way we will do it before evaluating your notebook!!!

Fill in your name and id number (numero matricola) below:

[1]:
```
NAME = "Tommaso Bergamasco"
ID_number = int("2052409")
# ID_number = int("0")

import IPython
assert IPython.version_info[0] >= 3, "Your version of IPython is too old,␣
 ↪please update it."
```

0.2 HOMEWORK #3

0.2.1 Non linear models for classification

In this notebook we are going to explore the use of SVM and Neural Networks for image classification. We are going to use the famous MNIST dataset, that is a dataset of handwritten digits. We get the data from mldata.org, that is a public repository for machine learning data.

[2]:
```
# Load the required packages
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sn
```

```python
import sklearn
from sklearn.datasets import fetch_openml
from sklearn.neural_network import MLPClassifier
```

In the following cell we will import the MNIST dataset. The `fetch_openml` function works differently depending on your version of scikit-learn. In particular, if you have the version 0.24 or higher, you need to add the argument `as_frame` and set it to `False`. Choose the correct line below (you can also try both of them: the wrong one will generate an error).

[3]:
```python
#load the MNIST dataset and let's normalize the features so that each value is␣
↪in [0,1]
## Load data from https://www.openml.org/d/554 - may take some time

# YOUR CODE HERE
# CHOOSE BETWEEN:
X, Y = fetch_openml('mnist_784', version=1, return_X_y=True, as_frame=False)
# OR:
# X, Y = fetch_openml('mnist_784', version=1, return_X_y=True)

print(f'The matrix X of the data has shape: {X.shape}')
print(f'Each image is represented as vector of shape {X[0].shape}')
print(f'The image is represented in gray scale levels {X[0]}')
print(f'Here it is a label: {Y[0]}')

# Rescale the data
X = X / 255.
```

```
The matrix X of the data has shape: (70000, 784)
Each image is represented as vector of shape (784,)
The image is represented in gray scale levels [  0.   0.   0.   0.   0.   0.
0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   3.  18.
  18.  18. 126. 136. 175.  26. 166. 255. 247. 127.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.  30.  36.  94. 154. 170. 253.
 253. 253. 253. 253. 225. 172. 253. 242. 195.  64.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.  49. 238. 253. 253. 253. 253. 253.
 253. 253. 253. 251.  93.  82.  82.  56.  39.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.  18. 219. 253. 253. 253. 253. 253.
```

```
 198. 182. 247. 241.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.  80. 156. 107. 253. 253. 205.
  11.   0.  43. 154.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.  14.   1. 154. 253.  90.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0. 139. 253. 190.
   2.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  11. 190. 253.
  70.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  35. 241.
 225. 160. 108.   1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  81.
 240. 253. 253. 119.  25.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  45. 186. 253. 253. 150.  27.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.  16.  93. 252. 253. 187.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0. 249. 253. 249.  64.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  46. 130. 183. 253. 253. 207.   2.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  39. 148.
 229. 253. 253. 253. 250. 182.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  24. 114. 221. 253.
 253. 253. 253. 201.  78.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.  23.  66. 213. 253. 253. 253.
 253. 198.  81.   2.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.  18. 171. 219. 253. 253. 253. 253. 195.
  80.   9.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.  55. 172. 226. 253. 253. 253. 253. 244. 133.  11.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0. 136. 253. 253. 253. 212. 135. 132.  16.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
Here it is a label: 5
```

[ ]:

In a classification problem it is desirable to split the dataset into train and test sets in a way that preserves the same proportions of examples in each class as observed in the original dataset, so that we work with *balanced* datasets. We can achieve this by setting the "stratify" argument of the function "train_test_split" to the Y component of our dataset.

We are going to use 500 samples in the train dataset, the remaining ones are used for testing.

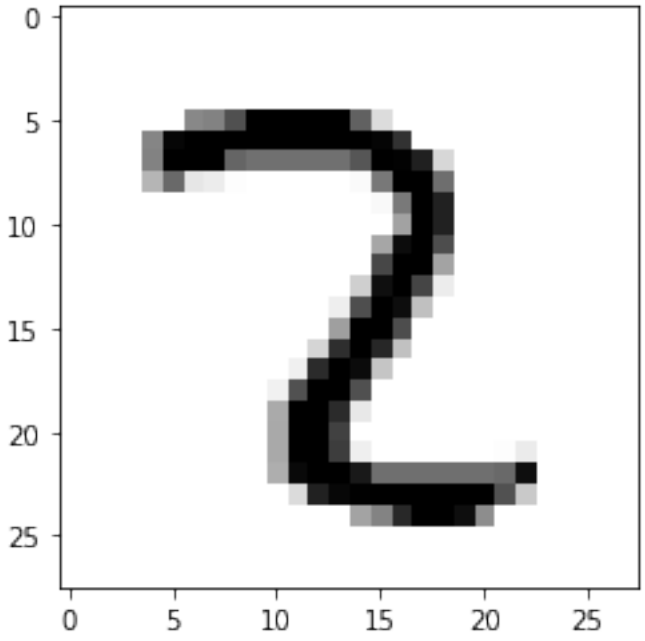
[4]:
```
from sklearn.model_selection import train_test_split

m_t = 500
x_train, x_test, y_train, y_test = train_test_split(X, Y, train_size=m_t/
 ↪len(Y), random_state=ID_number, stratify=Y)

print(f'Lenght train dataset: {len(y_train)}, Labels and frequencies: \n␣
 ↪{list(zip(*np.unique(y_train, return_counts=True)))}')
print(f'Lenght test dataset: {len(y_test)}, Labels and frequencies: \n␣
 ↪{list(zip(*np.unique(y_test, return_counts=True)))}')
```

```
Lenght train dataset: 500, Labels and frequencies:
 [('0', 49), ('1', 56), ('2', 50), ('3', 51), ('4', 49), ('5', 45), ('6', 49),
('7', 52), ('8', 49), ('9', 50)]
Lenght test dataset: 69500, Labels and frequencies:
 [('0', 6854), ('1', 7821), ('2', 6940), ('3', 7090), ('4', 6775), ('5', 6268),
('6', 6827), ('7', 7241), ('8', 6776), ('9', 6908)]
```

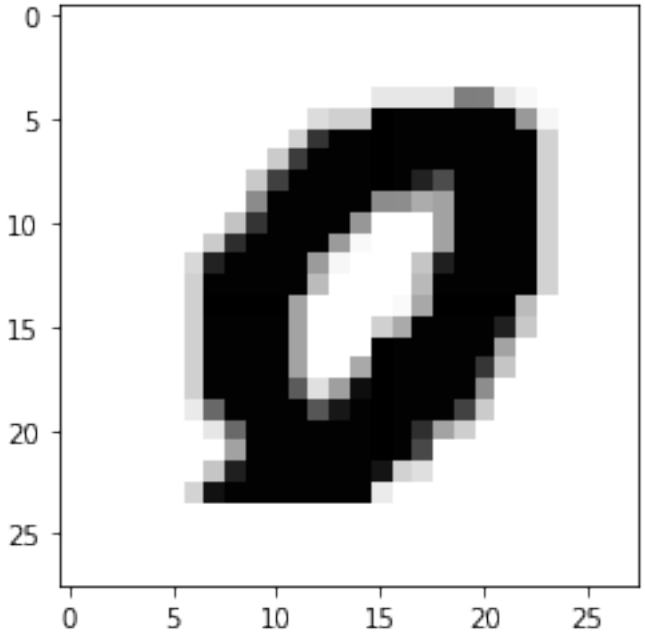
[5]:
```
# Function to plot a digit and print the corresponding label
def plot_digit(X_matrix, labels, index):
    print("INPUT:")
    plt.imshow(
        X_matrix[index].reshape(28,28),
        cmap          = plt.cm.gray_r,
        interpolation = "nearest"
    )
    plt.show()
    print(f"LABEL: {labels[index]}")
    return
```

[6]:
```
#let's try the plotting function
plot_digit(x_train, y_train, 100)
plot_digit(x_test, y_test, 40000)
```

```
INPUT:
```



LABEL: 2
INPUT:



LABEL: 0

0.3 TO DO 1

SVM with cross validation to pick the best model. Use SVC from sklearn.svm and GridSearchCV from sklearn.model_selection (5-fold cross-validation).

Print the best parameters found as well as the best score obtained by the 'optimal' model. Choose the grid: depending on the kernel you are using different hyper-parameters are needed (C, gamma, ...). You do not need to use more than 5 values for each hyper-parameter (otherwise the cell could be very slow).

[7]:
```
#import SVC
from sklearn.svm import SVC
#import for Cross-Validation
from sklearn.model_selection import GridSearchCV


def compute_best_SVM_with_CV(kernel_type : str, parameters : dict, x_train : np.
 ↪ndarray, y_train : np.ndarray) -> tuple:
    '''
    Use Cross validation to find the best SVM on the given parameters. Return␣
 ↪the best parameters set together with
    the corresponding score. Return also the scores for all the other␣
 ↪parameters given as input.
    :param kernel_type: Type of kernel (i.e. linear, rbf, poly)
    :param parameters: Dict containing kernel parameters (e.g. {'C': [1, 10,␣
 ↪100, 1000], 'gamma': [0.01, 0.001], ...})
    :param x_train: Train dataset
    :param y_train: Train labels

    :returns: (best_param, best_score, all_scores)
        WHERE:
        best_param: best parameter set (this is a dictionary)
        best_score: best score obtained for the given parameters (float)
        all_scores: all scores computed for each parameter (np.ndarray)
    '''
    SVM_model = SVC(kernel=kernel_type)
    # Use GridSearchCV to find the best parameter set.
    # YOUR CODE HERE

    # GridSearch clf (classifier) (5-fold CV by default)
    # The first argument must be an estimator, namely a sklearn model
    clf = GridSearchCV(SVM_model, parameters, return_train_score=True)
    clf.fit(x_train, y_train)

    print('#######################################')
    print(f'RESULTS for {kernel_type} KERNEL\n')
    # Store the best parameters set and print them
    print("Best parameters set found:")
```

```
    best_param = None
    # YOUR CODE HERE
    best_param = clf.best_params_
    print(best_param)

    # Store and print the score of the best parameters set
    print("\nScore with best parameters:")
    best_score = None
    # YOUR CODE HERE
    best_score = clf.best_score_
    print(best_score)

    # Store and print all the scores for the given parameters (average of the␣
 ↪validation scores)
    print("\nAll scores on the grid:")
    all_scores = None
    # YOUR CODE HERE
    results = clf.cv_results_
    all_scores = results['mean_test_score']
    print(all_scores)

    return best_param, best_score, all_scores

# Choose the grid for parameters of the linear SVM kernel
linear_parameters = None

# YOUR CODE HERE

# For C and gamma we consider exponentially growing sequences as suggested in:
# https://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf

linear_parameters = {'C': [0.01, 0.1, 1, 10, 100]}
best_param_lin, best_score_lin, all_scores_lin =␣
 ↪compute_best_SVM_with_CV('linear', linear_parameters, x_train, y_train)
# Choose the grid for parameters of the rbf SVM kernel
rbf_parameters = None

# YOUR CODE HERE
rbf_parameters = {'C': [0.01, 0.1, 1, 10, 100], 'gamma': [0.1, 0.01, 0.001]}
best_param_rbf, best_score_rbf, all_scores_rbf =␣
 ↪compute_best_SVM_with_CV('rbf', rbf_parameters, x_train, y_train)
# Choose the grid for parameters of the poly SVM kernel (do not forget to␣
 ↪choose the degree)
poly_parameters = None
# YOUR CODE HERE
poly_parameters = {'C': [0.01, 0.1, 1, 10, 100], 'gamma': [0.1, 0.01, 0.001],␣
 ↪'degree': [2, 3, 4, 5]}
```

```
best_param_poly, best_score_poly, all_scores_poly =␣
 ↪compute_best_SVM_with_CV('poly', poly_parameters, x_train, y_train)
```

```
####################################
RESULTS for linear KERNEL

Best parameters set found:
{'C': 0.01}

Score with best parameters:
0.8779999999999999

All scores on the grid:
[0.878 0.87  0.87  0.87  0.87 ]
####################################
RESULTS for rbf KERNEL

Best parameters set found:
{'C': 10, 'gamma': 0.01}

Score with best parameters:
0.898

All scores on the grid:
[0.112 0.112 0.112 0.112 0.472 0.112 0.674 0.888 0.732 0.698 0.898 0.898
 0.698 0.898 0.878]
####################################
RESULTS for poly KERNEL

Best parameters set found:
{'C': 0.1, 'degree': 2, 'gamma': 0.1}

Score with best parameters:
0.882

All scores on the grid:
[0.862 0.112 0.112 0.834 0.112 0.112 0.772 0.112 0.112 0.71  0.116 0.112
 0.882 0.334 0.112 0.848 0.218 0.112 0.776 0.166 0.112 0.71  0.15  0.112
 0.882 0.862 0.112 0.848 0.756 0.112 0.776 0.61  0.112 0.71  0.462 0.112
 0.882 0.882 0.334 0.848 0.834 0.112 0.776 0.766 0.112 0.71  0.652 0.112
 0.882 0.882 0.862 0.848 0.848 0.218 0.776 0.772 0.112 0.71  0.704 0.112]
```

[8]:
```
assert type(best_param_rbf) == dict
assert type(best_score_rbf) == np.float64
assert np.prod(np.array([len(params) for params in rbf_parameters.values()]))␣
 ↪== len(all_scores_rbf)
```

[9]:
```
# TODO 2:
# Get training and test error for the best SVM model obtained from CV (you need␣
 ↪to choose across different kernels
# too). You just need to look at the best model for each kernel and choose the␣
 ↪best one (you can do this by hand).

best_kernel_type, best_parameters = None, None
# YOUR CODE HERE

# Looking at the TODO 1 the best Kernel seems 'rbf'
# With best params: {'C': 10, 'gamma': 0.01}
best_kernel_type = 'rbf'
best_parameters = {'C': 10, 'gamma': 0.01}

best_SVM = SVC(kernel=best_kernel_type, **best_parameters).fit(x_train,y_train)
# best_SVM.fit(x_train, y_train)

# Compute training and test error for this model (use the usual sklearn␣
 ↪built-in functions seen in previous homeworks)
training_error, test_error = None, None
# YOUR CODE HERE

# Use the score function. (Error = 1 - score)
training_error = 1 - best_SVM.score(x_train, y_train)
test_error = 1 - best_SVM.score(x_test, y_test)

print (f"Best SVM training error: {training_error}")
print (f"Best SVM test error: {test_error}")
```

```
Best SVM training error: 0.0
Best SVM test error: 0.10402877697841728
```

[10]:
```
assert type(training_error) == np.float64
assert type(test_error) == np.float64
```

0.3.1 TO DO 3

Now we use feed-forward neural networks for classification. You can use the Multi-Layer-Perceptron (the multi-layer structure we have seen in class, see http://scikit-learn.org/stable/modules/neural_networks_supervised.html).

Note that we fix the starting random state so to make the runs reproducible. Use max_iter=1000, alpha=1e-4, solver='sgd', tol=1e-4, learning_rate_init=.1. Pick few architectures and use the default activation function (ReLU).

[11]:
```
# test different architectures:
# - 1 hidden layer with 10 nodes,
# - 1 hidden layer with 50 nodes,
```

```
# - 2 hidden layer with 10 nodes each,
# - 2 hidden layer with 50 nodes each
# feel free to change this and test more/different structures

# original dictionar of parameters
parameters = {'hidden_layer_sizes': [(10,), (50,), (10,10,), (50,50,)]}

# Experiments with NN (best params = (50,30,)) but a lot slower and score = 0.
 ↪874
# instead of score = 0.868 in the original NN
#parameters = {'hidden_layer_sizes': [(50,), (100,), (50,30,), (100,50,30,)]}

mlp = MLPClassifier(max_iter=1000, alpha=1e-4, solver='sgd', tol=1e-4,␣
 ↪random_state=ID_number, learning_rate_init=.1)

# Use GridSearchCV to find the best mlp using 5 fold CV.
mlp_CV = None
# YOUR CODE HERE

# Create GridSearchCV Object
mlp_CV = GridSearchCV(mlp, parameters).fit(x_train,y_train)

print('#####################################')
print ('RESULTS FOR NN\n')
# Store the best parameters set and print them
print("Best parameters set found:")
mlp_best_param = None
# YOUR CODE HERE
mlp_best_param = mlp_CV.best_params_
print(mlp_best_param)

# Store and print the score of the best parameters set
print("\nScore with best parameters:")
mlp_best_score = None
# YOUR CODE HERE
mlp_best_score = mlp_CV.best_score_
print(mlp_best_score)

# Store and print all the scores for the given parameters (average of the␣
 ↪validation scores)
print("\nAll scores on the grid:")
mlp_all_scores = None
# YOUR CODE HERE
results = mlp_CV.cv_results_
mlp_all_scores = results['mean_test_score']
print(mlp_all_scores)
```

```
####################################
RESULTS FOR NN

Best parameters set found:
{'hidden_layer_sizes': (50,)}

Score with best parameters:
0.868

All scores on the grid:
[0.832 0.868 0.806 0.85 ]
```

[12]:
```
assert type(mlp_best_param) == dict
assert type(mlp_best_score) == np.float64
```

0.4 TO DO 4

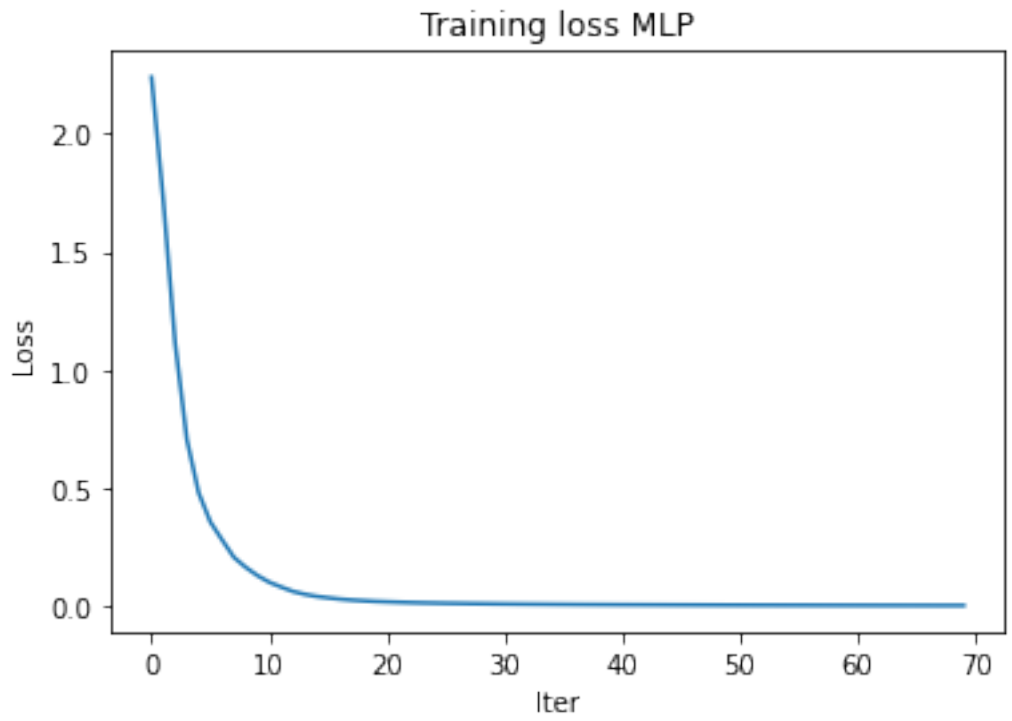
Now get training and test error for a NN with best parameters from above. We use verbose=True in input so to see how loss changes in iterations (see how this changes if the number of iterations is changed)

[13]:
```
# Get training and test error for the best NN model found using CV
max_iter = 1000

mlp = MLPClassifier(**mlp_best_param, max_iter=max_iter, alpha=1e-4,␣
 ↪solver='sgd', tol=1e-4, random_state=ID_number,
                    learning_rate_init=.1, verbose=True)

# ADD CODE: FIT MODEL & COMPUTE TRAINING AND TEST ERRORS
training_error, test_error = None, None
# YOUR CODE HERE

# Fit the model mlp
mlp = mlp.fit(x_train,y_train)

# Compute the Errors as (1 - score)
training_error = 1 - mlp.score(x_train, y_train)

test_error = 1 - mlp.score(x_test, y_test)

print ('\nRESULTS FOR BEST NN\n')

print ("Best NN training error: %f" % training_error)
print ("Best NN test error: %f" % test_error)
plt.plot(mlp.loss_curve_, label='Training Loss')
plt.title('Training loss MLP')
plt.xlabel('Iter'), plt.ylabel('Loss')
```

```
Iteration 1, loss = 2.23887816
Iteration 2, loss = 1.71824323
Iteration 3, loss = 1.11600659
Iteration 4, loss = 0.70669692
Iteration 5, loss = 0.48112168
Iteration 6, loss = 0.35693022
Iteration 7, loss = 0.28083475
Iteration 8, loss = 0.20758097
Iteration 9, loss = 0.16538020
Iteration 10, loss = 0.13035953
Iteration 11, loss = 0.10225928
Iteration 12, loss = 0.08206484
Iteration 13, loss = 0.06356067
Iteration 14, loss = 0.05128990
Iteration 15, loss = 0.04237660
Iteration 16, loss = 0.03619953
Iteration 17, loss = 0.03079414
Iteration 18, loss = 0.02685053
Iteration 19, loss = 0.02379731
Iteration 20, loss = 0.02107019
Iteration 21, loss = 0.01925352
Iteration 22, loss = 0.01760352
Iteration 23, loss = 0.01611787
Iteration 24, loss = 0.01499596
Iteration 25, loss = 0.01409227
Iteration 26, loss = 0.01314982
Iteration 27, loss = 0.01254213
Iteration 28, loss = 0.01177231
Iteration 29, loss = 0.01123417
Iteration 30, loss = 0.01070567
Iteration 31, loss = 0.01020400
Iteration 32, loss = 0.00981650
Iteration 33, loss = 0.00937935
Iteration 34, loss = 0.00910937
Iteration 35, loss = 0.00870309
Iteration 36, loss = 0.00840201
Iteration 37, loss = 0.00808977
Iteration 38, loss = 0.00784508
Iteration 39, loss = 0.00762300
Iteration 40, loss = 0.00734874
Iteration 41, loss = 0.00713611
Iteration 42, loss = 0.00696630
Iteration 43, loss = 0.00675293
Iteration 44, loss = 0.00658680
Iteration 45, loss = 0.00639718
Iteration 46, loss = 0.00623293
Iteration 47, loss = 0.00607951
Iteration 48, loss = 0.00592129
```

```
Iteration 49, loss = 0.00579565
Iteration 50, loss = 0.00563603
Iteration 51, loss = 0.00550819
Iteration 52, loss = 0.00538899
Iteration 53, loss = 0.00527600
Iteration 54, loss = 0.00517352
Iteration 55, loss = 0.00504697
Iteration 56, loss = 0.00494202
Iteration 57, loss = 0.00485409
Iteration 58, loss = 0.00475181
Iteration 59, loss = 0.00465091
Iteration 60, loss = 0.00456369
Iteration 61, loss = 0.00447551
Iteration 62, loss = 0.00439404
Iteration 63, loss = 0.00430955
Iteration 64, loss = 0.00423618
Iteration 65, loss = 0.00416096
Iteration 66, loss = 0.00409390
Iteration 67, loss = 0.00401210
Iteration 68, loss = 0.00395344
Iteration 69, loss = 0.00388256
Iteration 70, loss = 0.00382292
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs.
Stopping.

RESULTS FOR BEST NN

Best NN training error: 0.000000
Best NN test error: 0.142691
```

[13]: (Text(0.5, 0, 'Iter'), Text(0, 0.5, 'Loss'))



[14]:
```
assert type(training_error) == np.float64
assert type(test_error) == np.float64
```

0.5 TO DO 5

Write a function to find and plot the first digit (in x_test) that is missclassified by NN and correctly classified by SVM.

Write a function to compute the confusion matrix for the predictions of a model (on testset). If you are not familiar with what a confusion matrix is, have a look at this link: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html . You are not allowed to use sklearn to create the confusion matrix BUT you can compare your solution with the sklearn implementation to check you wrote it right (see assert checks).

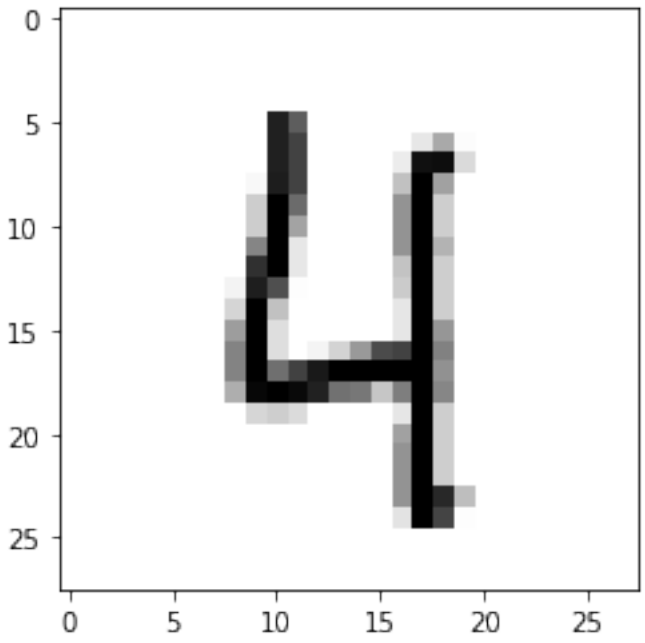
[15]:
```
def find_and_print_first_mismatched_prediction(SVM_prediction : np.ndarray,␣
↪NN_prediction : np.ndarray,
                                               x_test : np.ndarray, y_test : np.
↪ndarray) -> int:
    '''
    Function to find and print the first digit that is missclassified by NN and␣
↪correctly classified by SVM.
    :param SVM_prediction: SVM predicitons.
    :param NN_prediction: MLP predicitons.
    :param x_test: Test set inputs.
```

```
    :param y_test: Test set labels.

    :returns:
        i: returns the first index in which there is a mismatch between␣
 ↪NN_prediction and true labels but no mismatch
           between SVM_prediction and true labels.
    '''
    i = 0
    found = False
    while ((not found) and (i<len(y_test))):
        # YOUR CODE HERE
        # Case in witch we find the index and plot the corresponding digit
        if NN_prediction[i] != y_test[i] and SVM_prediction[i] == y_test[i]:
            found = True
            plot_digit(x_test, y_test, i)
            break
        else:
            i += 1
    return i


def confusion_matrix_by_hand(true_labels : np.ndarray, predicted_labels : np.
 ↪ndarray) -> np.ndarray:
    '''
    Function used to compute the confusion matrix given true and predicted␣
 ↪labels.
    :param true_labels: True labels.
    :param predicted_labels: Predicted labels (note this function does not␣
 ↪require to know which model generated
                              the predictions).

    :returns:
        confusion_matrix: Confusion matrix for the given true and predicted␣
 ↪labels.
    '''
    labels = np.unique(true_labels) # returns sorted unique values
    map_labels_to_index = {label:i for i, label in enumerate(labels)}
    confusion_matrix = np.zeros((len(labels), len(labels)))
    # YOUR CODE HERE

    for label in map_labels_to_index.keys():
        # extract the indeces in true_labels corresponding exactly to
        # the label of the current iteration
        indeces = np.nonzero(true_labels == label)[0]
        # Using the dict convert the true_labels and the predicted_labels at␣
 ↪the given
```

```
        # indeces in numbers instead of strings representing numbers. Doing␣
↪this allows us
        # to make numerical opperations between arrays.
        true_labels_in_numbers = np.array([map_labels_to_index[k] for k in␣
↪true_labels[indeces]])
        predicted_labels_in_numbers = np.array([map_labels_to_index[k] for k in␣
↪predicted_labels[indeces]])
        # Define a "distance-array" as (true_labels_in_numbers -␣
↪predicted_labels_in_numbers)
        # but considering only the indeces just found above.
        distance_array = true_labels_in_numbers - predicted_labels_in_numbers
        # if we are at the iteration e.g. label=5 such array will be composed␣
↪by integers in
        # the range [5-9=-4,5-0=5]. where -4 is obtained if we predict 9␣
↪instead of 5
        # (this will increment C_{5,9}) and 5 is obtained if we predict 0␣
↪instead of 5 (this
        # will increment C_{5,0}).
        # Using np.unique we obtain a sorted array as [-4,-3,...,4,5] (not␣
↪important) and
        # the occurences of every value which are exactly the
        # values C_{5,9},C_{5,8},...,C{5,0} of the confusion matrix.
        confusion_unique, confusion_counts = np.unique(distance_array,␣
↪return_counts=True)
        # flip the order:
        confusion_unique, confusion_counts = np.flip(confusion_unique), np.
↪flip(confusion_counts)
        # Again if label=5 we have now 2 arrays like: [5,4,...,-4] and [#of5,...
↪,#of-4]
        # Now we populate the confusion_matrix's row corresponding to the␣
↪current label
        label_in_number = map_labels_to_index[label]
        # for loop which iterates at maximum 10 times
        for elem, count in zip(confusion_unique, confusion_counts):
            # nested for loop which iterates maximum 10 times
            for i in range(10):
                # check if C_{label,i} != 0 (namely if such "bucket" exists in␣
↪the
                # confusion_unique array). This is obtained noting the fact␣
↪that every
                # confusion_unique array corresponding to a label is such as:
                # [label-0,label-1,...,label-9].
                if elem == label_in_number - i:
                    # In this case update the corresponding value of the C.M.
                    confusion_matrix[label_in_number,i] = count
                    break
```

```
        # this double for loop iterates at maximum 100 times so it's not so␣
 ↪computational demanding

    return confusion_matrix.astype(int)

# Let's test our functions
SVM_prediction = best_SVM.predict(x_test)
NN_prediction = mlp.predict(x_test)

first_index = find_and_print_first_mismatched_prediction(SVM_prediction,␣
 ↪NN_prediction, x_test, y_test)

SVM_CM = confusion_matrix_by_hand(y_test, SVM_prediction)
MLP_CM = confusion_matrix_by_hand(y_test, NN_prediction)

##### MY CODE
SVM_sklearn = sklearn.metrics.confusion_matrix(y_test, SVM_prediction)
MLP_sklearn = sklearn.metrics.confusion_matrix(y_test, NN_prediction)
print(f'SVM_sklearn comparison (if all 0 then same as sklearn):
 ↪\n{SVM_sklearn-SVM_CM}')
print(f'MLP_sklearn comparison (if all 0 then same as sklearn):
 ↪\n{MLP_sklearn-MLP_CM}')

print(f'SVM confusion matrix:\n{SVM_CM}')
print(f'MLP confusion matrix:\n{MLP_CM}')

# Convert confusion matrices to pandas data frames
labels = np.unique(y_test)
SVM_CM_df = pd.DataFrame(SVM_CM, index = labels, columns = labels)
MLP_CM_df = pd.DataFrame(MLP_CM, index = labels, columns = labels)

# Plot confusion matrices
def cap(df):
    df_numpy = df.to_numpy(dtype=int, copy=True)
    np.fill_diagonal(df_numpy, np.zeros(df_numpy.shape[0]))
    return np.amax(df_numpy)
fig, axes = plt.subplots(1,2, figsize=(20,5))
sn.heatmap(SVM_CM_df, annot=True, ax=axes[0], cmap='rocket_r',␣
 ↪vmax=cap(SVM_CM_df)*2, fmt='d')
sn.heatmap(MLP_CM_df, annot=True, ax=axes[1], cmap='rocket_r',␣
 ↪vmax=cap(MLP_CM_df)*2, fmt='d')
axes[0].set_title('SVM'), axes[1].set_title('MLP')
# Opional line to plot a better table if you see first and last row halved
#[ax.set_yticks(list(range(len(labels)+1))) for ax in axes]
```

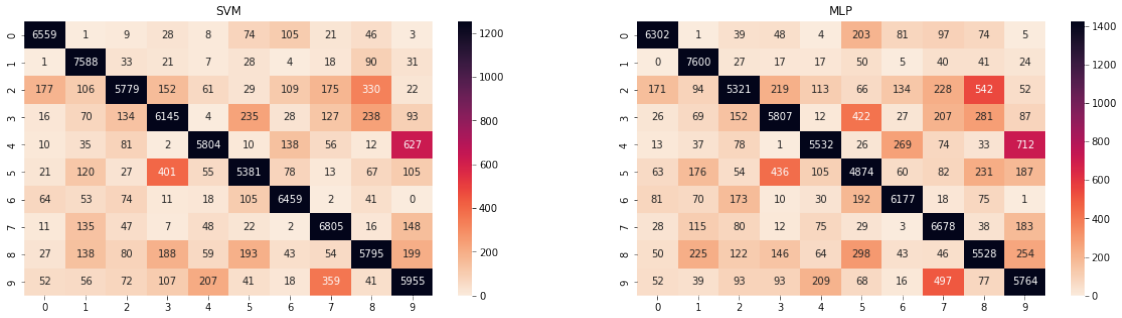
INPUT:




```
LABEL: 4
SVM_sklearn comparison (if all 0 then same as sklearn):
[[0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]]
MLP_sklearn comparison (if all 0 then same as sklearn):
[[0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]]
SVM confusion matrix:
[[6559    1    9   28    8   74  105   21   46    3]
 [   1 7588   33   21    7   28    4   18   90   31]
```

```
 [ 177  106 5779  152   61   29  109  175  330   22]
 [  16   70  134 6145    4  235   28  127  238   93]
 [  10   35   81    2 5804   10  138   56   12  627]
 [  21  120   27  401   55 5381   78   13   67  105]
 [  64   53   74   11   18  105 6459    2   41    0]
 [  11  135   47    7   48   22    2 6805   16  148]
 [  27  138   80  188   59  193   43   54 5795  199]
 [  52   56   72  107  207   41   18  359   41 5955]]
MLP confusion matrix:
[[6302    1   39   48    4  203   81   97   74    5]
 [   0 7600   27   17   17   50    5   40   41   24]
 [ 171   94 5321  219  113   66  134  228  542   52]
 [  26   69  152 5807   12  422   27  207  281   87]
 [  13   37   78    1 5532   26  269   74   33  712]
 [  63  176   54  436  105 4874   60   82  231  187]
 [  81   70  173   10   30  192 6177   18   75    1]
 [  28  115   80   12   75   29    3 6678   38  183]
 [  50  225  122  146   64  298   43   46 5528  254]
 [  52   39   93   93  209   68   16  497   77 5764]]
```

[15]: (Text(0.5, 1.0, 'SVM'), Text(0.5, 1.0, 'MLP'))



[16]:
```
from sklearn.metrics import confusion_matrix
skl_confusion_matrix_SVM = confusion_matrix(y_test, SVM_prediction)
skl_confusion_matrix_NN = confusion_matrix(y_test, NN_prediction)

assert np.sum(skl_confusion_matrix_SVM - SVM_CM) == 0
assert np.sum(skl_confusion_matrix_NN - MLP_CM) == 0
```

0.6 TO DO 6: explain the results you got (max 5 lines)

According to the cross-validation results, would you choose SVMs or NNs when 500 data points are available for training? Is this a good choice, given the results on the test set?

Looking at the confusion matrices what to do you observe? On which classes each model is more likely to make mistakes?

(Answer in the next cell, no need to add code)

0.7 # YOUR CODE HERE

- Given 500 training points the SVM performs better in terms of both Test error and execution time.
- We have $L_S^{SVM} = 0.0$ and $\hat{L}_D^{SVM} \simeq 0.104 \implies GAP \simeq 0.104$ which is quite high. Regularization `C` has already been tuned with CV so we need more Training Data.
- Given $C_{i,j} = \#$ of samples i classified as j, the more frequent missclassifications **for both the algorithms** are: $C_{5,3}, C_{3,5}, C_{9,7}, C_{2,8}, C_{4,9}$. Note also the asimmetry of the CMs ($C_{4,9} >> C_{9,4}; C_{2,8} >> C_{8,2}$, ecc.).

0.8 More Data

Now let's do the same but using more data points for training SVM and NN. For SVM we are going to use the best hyperparameters set (kernel, C, gamma, ...) found using 500 data points. For NNs we use the same best architecture as before, but you can try more if you want!

[17]:
```
#let restart the random generator with the given seed
np.random.seed(ID_number)

m_t = 60000
x_train, x_test, y_train, y_test = train_test_split(X, Y, train_size=m_t/
 ↪len(Y), random_state=ID_number, stratify=Y)

print(f'Lenght train dataset: {len(y_train)}, Labels and frequencies: \n␣
 ↪{list(zip(*np.unique(y_train, return_counts=True)))}')
print(f'Lenght test dataset: {len(y_test)}, Labels and frequencies: \n␣
 ↪{list(zip(*np.unique(y_test, return_counts=True)))}')
```

```
Lenght train dataset: 60000, Labels and frequencies:
 [('0', 5917), ('1', 6752), ('2', 5991), ('3', 6121), ('4', 5849), ('5', 5411),
('6', 5894), ('7', 6251), ('8', 5850), ('9', 5964)]
Lenght test dataset: 10000, Labels and frequencies:
 [('0', 986), ('1', 1125), ('2', 999), ('3', 1020), ('4', 975), ('5', 902),
('6', 982), ('7', 1042), ('8', 975), ('9', 994)]
```

[18]:
```
# As we did with the first HW let's use a decorator to measure time
from collections import defaultdict
running_times = defaultdict(list)

def measure_time(function):
    def wrap(*args, **kw):
        import time
        t_start = time.time()
        result = function(*args, **kw)
        t_end = time.time()
```

```
        running_times[type(args[0]).__name__].append(t_end - t_start)
        return result
    return wrap

@measure_time
def fit_classification_model(model, x_train, y_train):
    model.fit(x_train, y_train)
```

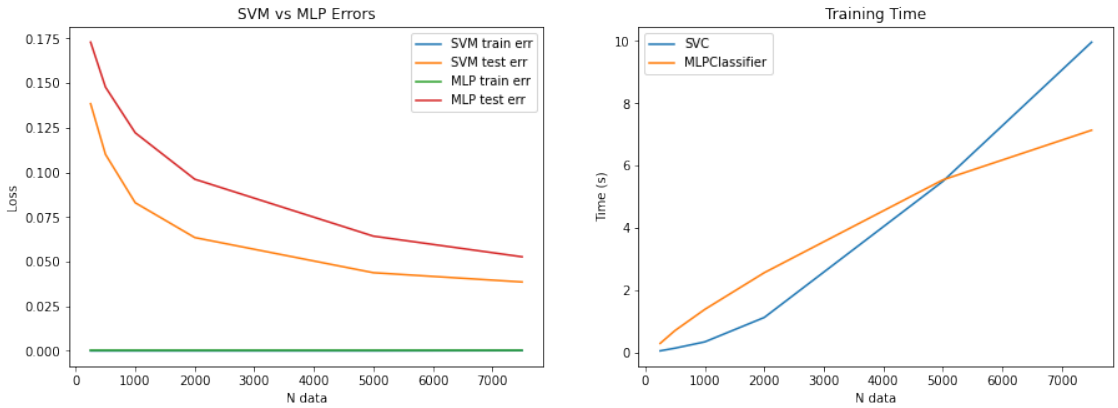
[19]:
```
n_data = [250, 500, 1000, 2000, 5000, 7500]
svm_train_err, svm_test_err = [], []
mlp_train_err, mlp_test_err = [], []
for n in n_data:
    print(f'Processing with {n} data ...')
    # Initialize models according to the best we got using 500 data
    svm = SVC(kernel=best_kernel_type, **best_parameters)
    mlp = MLPClassifier(**mlp_best_param, max_iter=max_iter, alpha=1e-4,␣
 ↪solver='sgd', tol=1e-4,
                        random_state=ID_number, learning_rate_init=.1)

    # fit svc
    fit_classification_model(svm, x_train[:n], y_train[:n])
    # get svc train and test error
    svm_train_err.append(1. - svm.score(x_train[:n], y_train[:n]))
    svm_test_err.append(1. - svm.score(x_test, y_test))

    # fit mlp
    fit_classification_model(mlp, x_train[:n], y_train[:n])
    # get mlp train and test error
    mlp_train_err.append(1. - mlp.score(x_train[:n], y_train[:n]))
    mlp_test_err.append(1. - mlp.score(x_test, y_test))
```

```
Processing with 250 data …
Processing with 500 data …
Processing with 1000 data …
Processing with 2000 data …
Processing with 5000 data …
Processing with 7500 data …
```

[20]:
```
fig, axes = plt.subplots(1,2, figsize=(15, 5))
axes[0].plot(n_data, np.array(svm_train_err), label='SVM train err')
axes[0].plot(n_data, np.array(svm_test_err), label='SVM test err')
axes[0].plot(n_data, np.array(mlp_train_err), label='MLP train err')
axes[0].plot(n_data, np.array(mlp_test_err), label='MLP test err')
axes[0].set_xlabel('N data'), axes[0].set_ylabel('Loss')
axes[0].legend(), axes[0].set_title('SVM vs MLP Errors')

for model, times in running_times.items():
```

```
    axes[1].plot(n_data, times, label=model)
axes[1].set_xlabel('N data'), axes[1].set_ylabel('Time (s)')
axes[1].legend(), axes[1].set_title('Training Time')
```

[20]: (<matplotlib.legend.Legend at 0x7f967eae2190>, Text(0.5, 1.0, 'Training Time'))



1 TODO 7: Complete dataset

In the following we will compare the last studied regression method (e.g. NN, with Multi-Layer Perceptron) with one from the very first homework.

Among the different linear classifiers, we choose logistic regression (with standard parameters from scikit-learn but the number of iteration), given the good performances we obtained some weeks ago.

[21]:
```
from sklearn import linear_model

# Fit and test a logistic regression model
max_iter = 1000
log_reg = None
training_error, test_error = None, None
# YOUR CODE HERE

# Define the logistic regression model
log_reg = linear_model.LogisticRegression(max_iter=max_iter).
 ↪fit(x_train,y_train)

# training error
training_error_lr = 1 - log_reg.score(x_train, y_train)
# test error
test_error_lr = 1 - log_reg.score(x_test, y_test)
```

```
print (f"Best logistic regression training error: {training_error_lr:.4f}")
print (f"Best logistic regression test error: {test_error_lr:.4f}")
```

```
Best logistic regression training error: 0.0613
Best logistic regression test error: 0.0756
```

We now learn the NN. Below we use the same best architecture as before (found with 500 data), feel free to try larger ones (and to use again CV), or smaller ones if it takes too much time. (We suggest that you use 'verbose=True' so have an idea of how long it takes to run 1 iteration).

Note: If you do again CV to choose the best architecture remember to save the best set of parameters into the variable: "mlp_best_param".

[22]:
```
#get training and test error for the best NN model from CV
best_mlp_large = None
training_error, test_error = None, None
# YOUR CODE HERE

# CODE TO REPEAT CROSS VALIDATION (TOO SLOW)
# parameters = {'hidden_layer_sizes': [(10,), (30,), (50,)]}
# mlp = MLPClassifier(max_iter=max_iter, alpha=1e-4, solver='sgd', tol=1e-4,␣
 ↪random_state=ID_number, learning_rate_init=.1)
# mlp_CV = GridSearchCV(mlp, parameters).fit(x_train,y_train)
# mlp_best_param = mlp_CV.best_params_
# print(mlp_best_param)

# fit the model using same parameters as before
best_mlp_large = sklearn.neural_network.MLPClassifier(**mlp_best_param,␣
 ↪max_iter=max_iter, alpha=1e-4, solver='sgd', tol=1e-4,␣
 ↪random_state=ID_number,learning_rate_init=.1, verbose=True).
 ↪fit(x_train,y_train)

# training error
training_error = 1 - best_mlp_large.score(x_train, y_train)
#test error
test_error = 1 - best_mlp_large.score(x_test, y_test)

print ('\nRESULTS FOR BEST NN\n')

print (f"Best NN training error: {training_error:.4f}")
print (f"Best NN test error: {test_error:.4f}")
```

```
Iteration 1, loss = 0.31444007
Iteration 2, loss = 0.14321969
Iteration 3, loss = 0.10684830
Iteration 4, loss = 0.08836894
Iteration 5, loss = 0.07452152
Iteration 6, loss = 0.06457757
```

```
Iteration 7, loss = 0.05926571
Iteration 8, loss = 0.05053191
Iteration 9, loss = 0.04544237
Iteration 10, loss = 0.04080749
Iteration 11, loss = 0.03767561
Iteration 12, loss = 0.03430917
Iteration 13, loss = 0.03107942
Iteration 14, loss = 0.02855702
Iteration 15, loss = 0.02678904
Iteration 16, loss = 0.02226507
Iteration 17, loss = 0.02080039
Iteration 18, loss = 0.01817122
Iteration 19, loss = 0.01520793
Iteration 20, loss = 0.01492882
Iteration 21, loss = 0.01383594
Iteration 22, loss = 0.01173312
Iteration 23, loss = 0.01039332
Iteration 24, loss = 0.00942098
Iteration 25, loss = 0.00805712
Iteration 26, loss = 0.00734110
Iteration 27, loss = 0.00629117
Iteration 28, loss = 0.00552045
Iteration 29, loss = 0.00450284
Iteration 30, loss = 0.00393818
Iteration 31, loss = 0.00380169
Iteration 32, loss = 0.00348441
Iteration 33, loss = 0.00306153
Iteration 34, loss = 0.00287505
Iteration 35, loss = 0.00266855
Iteration 36, loss = 0.00225637
Iteration 37, loss = 0.00218406
Iteration 38, loss = 0.00210596
Iteration 39, loss = 0.00209815
Iteration 40, loss = 0.00189765
Iteration 41, loss = 0.00178029
Iteration 42, loss = 0.00172751
Iteration 43, loss = 0.00169954
Iteration 44, loss = 0.00161414
Iteration 45, loss = 0.00153779
Iteration 46, loss = 0.00150693
Iteration 47, loss = 0.00146289
Iteration 48, loss = 0.00141592
Iteration 49, loss = 0.00136872
Iteration 50, loss = 0.00135188
Iteration 51, loss = 0.00131967
Iteration 52, loss = 0.00129462
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs.
Stopping.
```

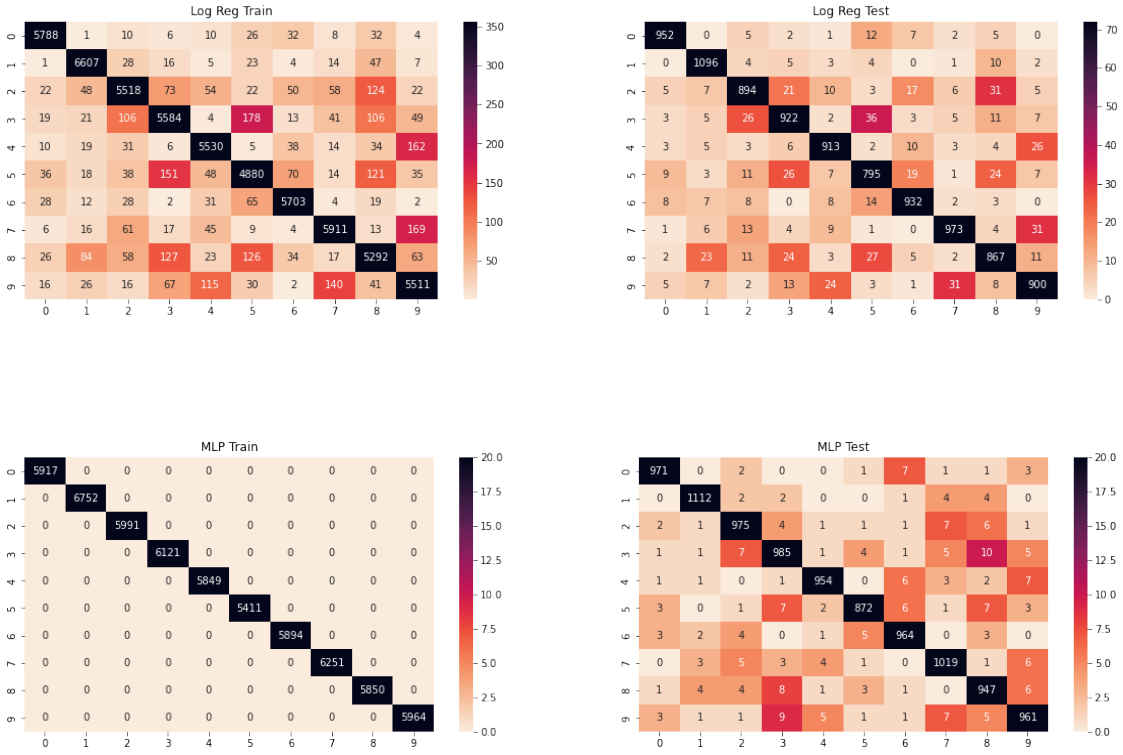
```
RESULTS FOR BEST NN

Best NN training error: 0.0000
Best NN test error: 0.0240
```

[23]:
```
assert type(training_error) == np.float64
assert type(test_error) == np.float64
```

[24]:
```
## TODO 8: compute the confusion matrices both on train and test set for␣
 ↪Logistic regression (trained on 60k)
# and MLP (trained on 60k).

# Log Reg Confusion matrices
log_reg_CM_train, log_reg_CM_test = None, None
# YOUR CODE HERE
log_reg_CM_train = confusion_matrix_by_hand(y_train, log_reg.predict(x_train))
log_reg_CM_test = confusion_matrix_by_hand(y_test, log_reg.predict(x_test))

# mlp
mlp_CM_train, mlp_CM_test = None, None
# YOUR CODE HERE
mlp_CM_train = confusion_matrix_by_hand(y_train, best_mlp_large.
 ↪predict(x_train))
mlp_CM_test = confusion_matrix_by_hand(y_test, best_mlp_large.predict(x_test))

# Convert confusion matrices to pandas data frames
labels = np.unique(y_test)
log_reg_CM_train_df = pd.DataFrame(log_reg_CM_train, index = labels, columns =␣
 ↪labels)
log_reg_CM_test_df = pd.DataFrame(log_reg_CM_test, index = labels, columns =␣
 ↪labels)

mlp_CM_train_df = pd.DataFrame(mlp_CM_train, index = labels, columns = labels)
mlp_CM_test_df = pd.DataFrame(mlp_CM_test, index = labels, columns = labels)

# Plot confusion matrices
fig, axes = plt.subplots(1,2, figsize=(20,5))
sn.heatmap(log_reg_CM_train_df, annot=True, ax=axes[0], cmap='rocket_r',␣
 ↪vmax=cap(log_reg_CM_train_df)*2, fmt='d')
sn.heatmap(log_reg_CM_test_df, annot=True, ax=axes[1], cmap='rocket_r',␣
 ↪vmax=cap(log_reg_CM_test_df)*2, fmt='d')
axes[0].set_title('Log Reg Train'), axes[1].set_title('Log Reg Test')
# Opional line to plot a better table if you see first and last row halved
# [ax.set_yticks(list(range(len(labels)+1))) for ax in axes]

fig, axes = plt.subplots(1,2, figsize=(20,5))
```

```
sn.heatmap(mlp_CM_train_df, annot=True, ax=axes[0], cmap='rocket_r',␣
 ↪vmax=cap(mlp_CM_test_df)*2, fmt='d')
sn.heatmap(mlp_CM_test_df, annot=True, ax=axes[1], cmap='rocket_r',␣
 ↪vmax=cap(mlp_CM_test_df)*2, fmt='d')
axes[0].set_title('MLP Train'), axes[1].set_title('MLP Test')
# Opional line to plot a better table if you see first and last row halved
# [ax.set_yticks(list(range(len(labels)+1))) for ax in axes]
```

[24]: (Text(0.5, 1.0, 'MLP Train'), Text(0.5, 1.0, 'MLP Test'))



[25]:
```
assert log_reg_CM_train.shape == (10, 10)
assert log_reg_CM_test.shape == (10, 10)
assert mlp_CM_train.shape == (10, 10)
assert mlp_CM_test.shape == (10, 10)
```

1.1 TO DO 9

Compare and discuss: - compare the computational time required to fit a SVM and a MLP. Which is faster as the number of data increase? Why? Can you apply both methods in the high data regime? - the results from SVM m=7500 and NN with m=60000 training data points. - the results from NN with m=500 and m=60000 training data points. - What do you observe in the confusion matrices? Which are the hardest classes? Are the hardest and easiest classes the same both for mlp and logistic regression?

(Answer in the next cell, no need to write code)

1.2 # YOUR CODE HERE

####
Answer
1 +
Look-
ing at
the plot
just
before
the
TODO7
we can
notice
that
the
SVM is
faster
for
smaller
data
sets.
When
the
data
set size
in-
creases
(in par-
ticular
in cor-
respon-
dence
of $m \simeq$
5000)
we see
that
the
MLP
starts
to
behave
better.
#####
Com-
plexity
of SVM
+ The
com-
plexity
of the
SVM
(look-
ing at
the

Answer 1 + Looking at the plot just before the TODO7 we can notice that the SVM is faster for smaller data sets. When the data set size increases (in particular in correspondence of $m \simeq 5000$) we see that the MLP starts to behave better.

Complexity of SVM + The complexity of the SVM (looking at the

Answer 3

- Looking at the Test Errors we have: $\hat{L}^{NN}_{D_{500}} = 0.142691$ and $\hat{L}^{NN}_{D_{60000}} = 0.024$ (reduction of $\hat{L}_D$ of a factor $\simeq 6$).
- The Training Errors L_S are in both cases 0. While in the first case we probably have some overfitting, in the second case, as said above, the Generalization Gap is "small enough" to suggest overfitting is not a problem.
-

1.3 A seemingly counterintuitive result is that when $m = 500$ the NN stops after 70 iterations, when $m = 60000$ it stops after only 52 iterations. This means that in the second case the $L_S(\vartheta)$ function is minimized "faster" by the SGD solver. This is probably due to the fact that when using more data the function to be minimized is more regular (less local minima) and the SGD performs better.

- The first fact to note is the difference between the 2 **Training CM** where the Log. Reg. one is the only matrix which does classification errors. This is due to the fact that the Log. Reg.'s decision boundaries are just lines and cannnot perfectly fit a Non-Linearly Separable data set, while the MLP is able to fit also Non-Linear models.
- Note also how the CM for the models trained on 60000 points are more symmetrical than the ones trained only on 500 points (meaning that in the former case a '3' is classified as a '5' more or less as much times a '5' is classified as a '3': $C_{3,5} \simeq C_{5,3}$). This is clearly a more intuitive behaviour and it's probably due to the "quality" of the predicted model (better model when a lot of data are available for Training).
- Following the same notation for the Confusion Matrices used above, where:

  $$C_{i,j} = \#\text{ of samples i classified as j}$$

  The most difficult classes could be e.g.

  - For Log. Reg.: $C_{3,5}; C_{4,9}; C_{5,3}; C_{7,9}; C_{9,7}$
  - For NN: $C_{0,6}; C_{3,8}; C_{8,3}; C_{9,3}$
- This selection of "hard classes" is not exaustive and selecting some more of them would show that LR and NN confuse often the same classes. The fact that the "hardest classes" are not in common can be attributed to 2 main facts:

  1. The NN does a significantly better job in the test set, leading to a lot less missclassifications than the LR. This means that what we call "confusion" in the NN could even be simply "random noise" (equivalent to think that missclassifications with MLP are more evenly distributed).
  2. **LR and NN** learn differently: the former learns linear boundaries, the latter (in this case) learns Non-Linear ones. This explanation is also supported by the fact that the CM of the **SVM and NN** (both used for Non-Linear models) with 500 training data (see 15-th cell) are on the contrary very similar to each other in terms of "hardest classes".

- **1.4 Given a classification problem for LR the easiest scenario takes place when data are linearly separable and I can simply draw a line which divides the classes. Given a Non-Linear classification problem for MLP the easiest scenario takes place when I can divide data with the "simplest model" which is a line as in the LR case. For this reason (when we don't have overfitting) the "easiest classes" for LR and NN should be similar. Looking at the CM this is the case.**
