Hand_Digit_with_NN_professor

November 11, 2022

Table of Contents

1 MNIST Hand Digit Classification

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

size=18
params = {'legend.fontsize': 'Large',
          'axes.labelsize': size,
          'axes.titlesize': size,
          'xtick.labelsize': size*0.75,
          'ytick.labelsize': size*0.75}
plt.rcParams.update(params)
```

In this notebook we will explore different strategies for solving a classification problem consisting of determining the handwritten digit corresponding to a $ 28 \times 28 $ pixel image (MNIST dataset).

- We will start by tackling a binary classification problem using different classification algorithms whose implementations are available in scikit-learn, including the multilayer perceptron as an example of a multilayer type neural network (multilayer perceptron, MLP).
- Next, we will consider multiclass classification, and calculate the performance of the previous algorithms in this more challenging problem.

- The last part of the notebook contains an introduction to PyTorch, one of the most widely used libraries for neural network training. We will review the basic concepts of Pytorch and the different modules that allow simplifying the implementation and optimization of neural networks, including the design of convolutional neural networks (CNNs) that represent a more powerful alternative than MLPs in image classification problems.

For faster executions you must run PyTorch code on GPU. For this, it is sufficient to use Gooble Colab by properly configuring the runtime environment.

2 Part 1. Scikit-learn Methods

2.1 1. Data Preparation

2.1.1 1.1. Data load

MNIST is a handwritten digit classification problem that includes 60,000 training patterns and 10,000 test patterns, with representations obtained from the digitization of the corresponding grayscale images and with 28×28 pixel resolution.

The database can be downloaded from the OpenML repository using tools available from Scikit-learn. It is recommended that after a first download you make a local copy of the data to speed up future executions.

[2]:
```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

# Load data from https://www.openml.org/d/554
X, y = fetch_openml("mnist_784", version=1, return_X_y=True, as_frame=False)
# Format data in y as integers
y = y.astype(np.int)

print('Size of Input Data Matrix:', X.shape)
print('Size of Label Vector:', y.shape)
```

```
Size of Input Data Matrix: (70000, 784)
Size of Label Vector: (70000,)
```

```
/var/folders/yx/m14y5d9x0658rc8q1p_hplc000022_/T/ipykernel_50968/4184569487.py:7
: DeprecationWarning: `np.int` is a deprecated alias for the builtin `int`. To
silence this warning, use `int` by itself. Doing this will not modify any
behavior and is safe. When replacing `np.int`, you may wish to use e.g.
`np.int64` or `np.int32` to specify the precision. If you wish to review your
current use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y = y.astype(np.int)
```

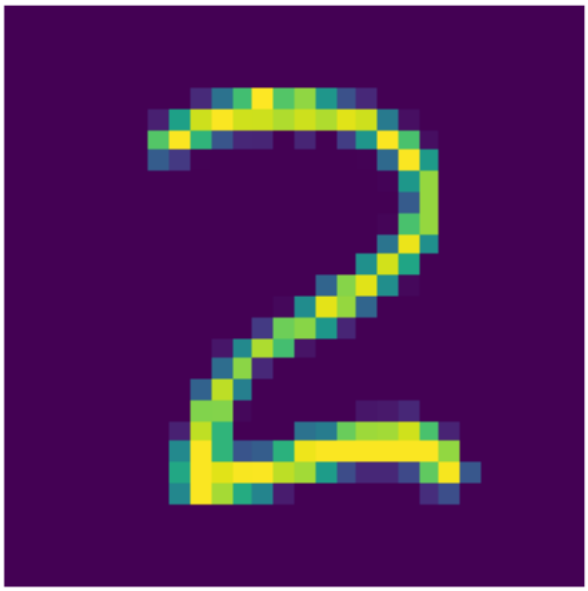
Exercise 1.1: Save data variables `X` and `y` so that you can use them in the future without donwloading the dataset again. Reload data from file to check the correctness of your solution. You can use `numpy` methods `savez` and `load` for this purpose.

[3]:
```
# You may use numpy.savez
# Reload variables with names Xlocal and ylocal

# <SOL>
np.savez('MNIST.npz', X=X, y=y)
data = np.load('MNIST.npz')
Xlocal = data['X']
ylocal = data['y']
# </SOL>

# Check that the reloaded matrices shapes are the same as before
print('Size of Input Data Matrix:', Xlocal.shape)
print('Size of Label Vector:', ylocal.shape)
```

```
Size of Input Data Matrix: (70000, 784)
Size of Label Vector: (70000,)
```

Each row of `X` contains a different digit. You can display the original images by realigning the dimensions of each row of `X`, as shown in the following example:

[4]:
```
pos = 1900

plt.imshow(X[pos,].reshape(28, 28)), plt.axis('off'), plt.show()
print(f'The label of element {pos} is {y[pos]}')
```



```
The label of element 1900 is 2
```

2.1.2 1.2. Data partitioning

To begin, we will perform a random partition of the available data into training and test sets of 60,000 and 10,000 digits, respectively.

The notebook also considers a binary problem consisting of the recognition of digits 7 and 9. We have selected this pair of digits as it is one of the most confusing, but if you wish, you can use any other pair of digits.

Exercise 1.2: Create the binary classification problem 7 vs 9. Save the corresponding data matrices with names `X_tr_bin`, `y_tr_bin`, `X_tst_bin`, `y_tst_bin`. Make sure that the target vectors contain just 0s (for class 7) and 1s (class 9).

[5]:
```
from sklearn.model_selection import train_test_split

train_samples = 60000
test_samples = 10000

X_tr, X_tst, y_tr, y_tst = train_test_split(
    Xlocal, ylocal, train_size=train_samples, test_size=test_samples,
    random_state=0)

print('Shape of input training data (multiclass):', X_tr.shape)
print('Shape of target training vector (multiclass):', y_tr.shape)
print('Shape of input test data (multiclass):', X_tst.shape)
print('Shape of target test vector (multiclass):', y_tst.shape)

# Exercise: create the binary classification problem 7 vs 9
# <SOL>
digit1 = 7
digit2 = 9

# Train data
X_tr_bin = X_tr[(y_tr == digit1) + (y_tr == digit2),]
y_tr_bin = y_tr[(y_tr == digit1) + (y_tr == digit2)]
y_tr_bin[y_tr_bin == digit1] = 0
y_tr_bin[y_tr_bin == digit2] = 1

# Repeat for test
X_tst_bin = X_tst[(y_tst == digit1) + (y_tst == digit2),]
y_tst_bin = y_tst[(y_tst == digit1) + (y_tst == digit2)]
y_tst_bin[y_tst_bin == digit1] = 0
y_tst_bin[y_tst_bin == digit2] = 1
# </SOL>

print('\nShape of input training data (binary):', X_tr_bin.shape)
print('Shape of target training vector (binary):', y_tr_bin.shape)
print('Shape of input test data (binary):', X_tst_bin.shape)
print('Shape of target test vector (binary):', y_tst_bin.shape)
```

```
Shape of input training data (multiclass): (60000, 784)
Shape of target training vector (multiclass): (60000,)
Shape of input test data (multiclass): (10000, 784)
Shape of target test vector (multiclass): (10000,)

Shape of input training data (binary): (12218, 784)
Shape of target training vector (binary): (12218,)
Shape of input test data (binary): (2033, 784)
Shape of target test vector (binary): (2033,)
```

1.3. Data normalization

Next we will normalize the data for the multiclass and binary problems. When working with images, it is frequent to normalize the input data, corresponding to the gray intensity values of the different pixels, so that they take values in the range $[-0.5, 0.5]$.

Exercise 1.3: Implement the normalization of the input data using the `MinMaxScaler` from `scikit-learn`. Make sure to normalize data independently for the multiclass and binary classification cases. Make also sure not to refit the scaler object when transforming the test partitions.

[6]:
```
# <SOL>
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(-.5, .5))
X_tr = scaler.fit_transform(X_tr)
X_tst = scaler.transform(X_tst)

X_tr_bin = scaler.fit_transform(X_tr_bin)
X_tst_bin = scaler.transform(X_tst_bin)
# </SOL>
```

2.2 2. Binary classification

2.2.1 2.1. Two dimensional representation

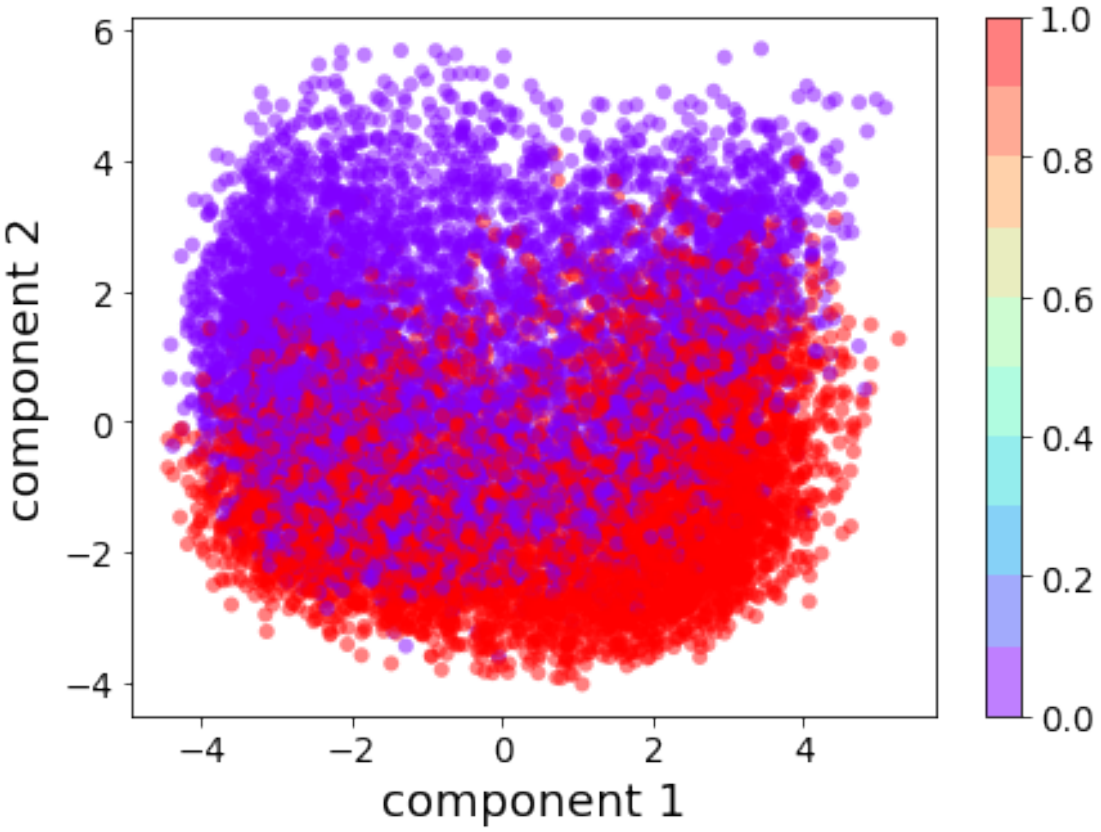
In order to be able to represent the classification frontier, we will begin our exploration by working on the two variables that present the greatest dispersion. To do this, we will use the Principal Component Analysis (PCA) algorithm.

2.1.1. Principal Component Analysis (PCA)

Exercise 2.1: Obtain the first two PCA projections for the binary classification problem. Store your results in the variables `X_tr_2D` and`X_tst_2D`. Make a scatter plot of these dimensions distinguishing the points corresponding to both classes, and reflect on the type of classification frontier that would provide a lower error rate.

[7]:
```
from sklearn.decomposition import PCA

# <SOL>
pca = PCA(2)  # project from 784 to 2 dimensions
X_tr_2D = pca.fit_transform(X_tr_bin)
```

```
X_tst_2D = pca.transform(X_tst_bin)
# </SOL>

plt.figure(figsize=(7,5))
plt.scatter(X_tr_2D[:, 0], X_tr_2D[:, 1],
            c=y_tr_bin, edgecolor='none', alpha=0.5,
            cmap=plt.cm.get_cmap('rainbow', 10))
plt.xlabel('component 1')
plt.ylabel('component 2')
plt.colorbar();
```



2.1.2. Linear Classification with Logistic Regression First we will analyze the behavior of logistic regression for this dataset **using just the two first dimensions of PCA as the input variables.**

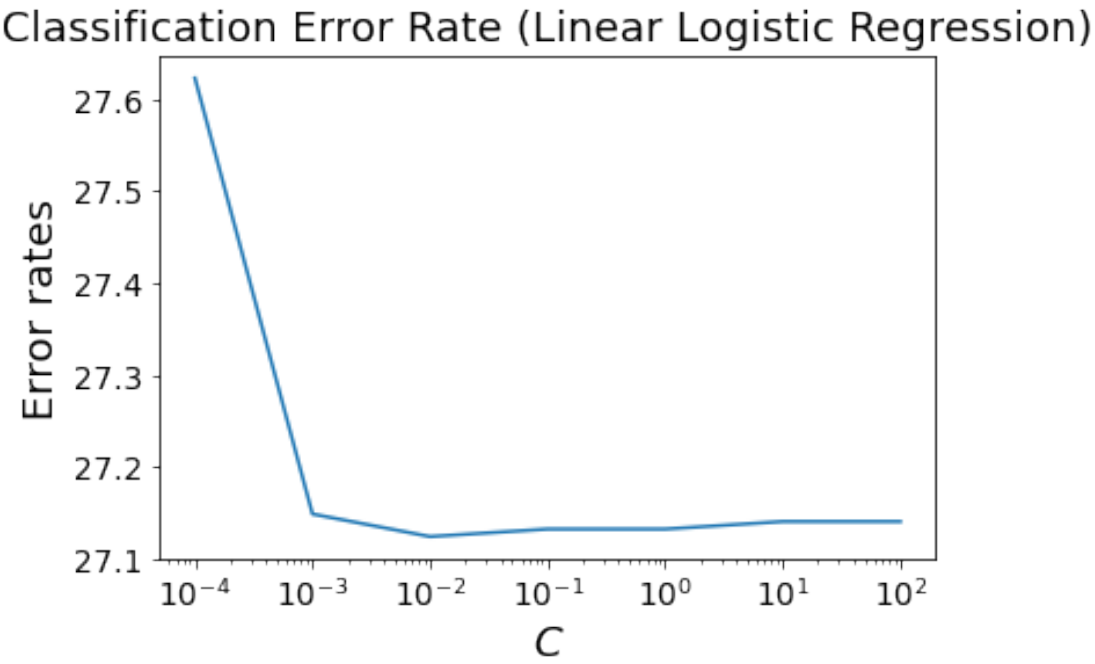
Exercise 2.2: Use `GridSearchCV` and `LogisticRegression` methods from `sklearn` to calculate the average classification error for different values of the inverse regularization parameter C. Explore C using a logarithmic scale, e.g.,

$$C = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001]$$

Use a 5-fold strategy to obtain the best value of parameter C.

Store the best classifier in object `clf` (otherwise, some of the next cells might fail).

[8]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Create an array with the explored values of $C$
# C_param = <FILL IN>
C_param = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001]

# Write the code to apply grid search here:
# <SOL>
parameters = {'C': C_param}
clf = GridSearchCV(LogisticRegression(max_iter=250, tol=1e-3), parameters, cv=5)
clf.fit(X_tr_2D, y_tr_bin)
# </SOL>

# Create and array with the validation error rates for the corresponding values␣
 ↪of C
# CE_param = <FILL IN>
CE_param = 100 - 100 * clf.cv_results_['mean_test_score']

plt.semilogx(C_param, CE_param)
plt.xlabel('$C$'), plt.ylabel('Error rates')
plt.title('Classification Error Rate (Linear Logistic Regression)')
plt.show()
```



Exercise 2.3: Calculate the average classification error rate (CE) and negative log-likelihood (NLL) of the best classifier.

[9]:
```
# Compute the average Classification Error Rate of previous classifier␣
 ↪calculated over the test data
# CE_LR2D = <FILL IN>
CE_LR2D = 100 - 100*np.mean(y_tst_bin==clf.predict(X_tst_2D))

# Compute probabilistic (or log-probabilistic) predictions
# q_pred = <FILL IN>
q_pred = clf.predict_log_proba(X_tst_2D)

# Compute the Negative log-lilelihood of previous classifier calculated over␣
 ↪the test data
# NLL_LR2D = <FILL IN>
NLL_LR2D = - np.sum(y_tst_bin.T @ q_pred[:,1]) - np.sum((1-y_tst_bin).T @␣
 ↪q_pred[:,0])

print('Error rate of Linear Logistic Regression Classifier (%):', CE_LR2D)
print('NLL of Linear Logistic Regression Classifier:', NLL_LR2D)
```

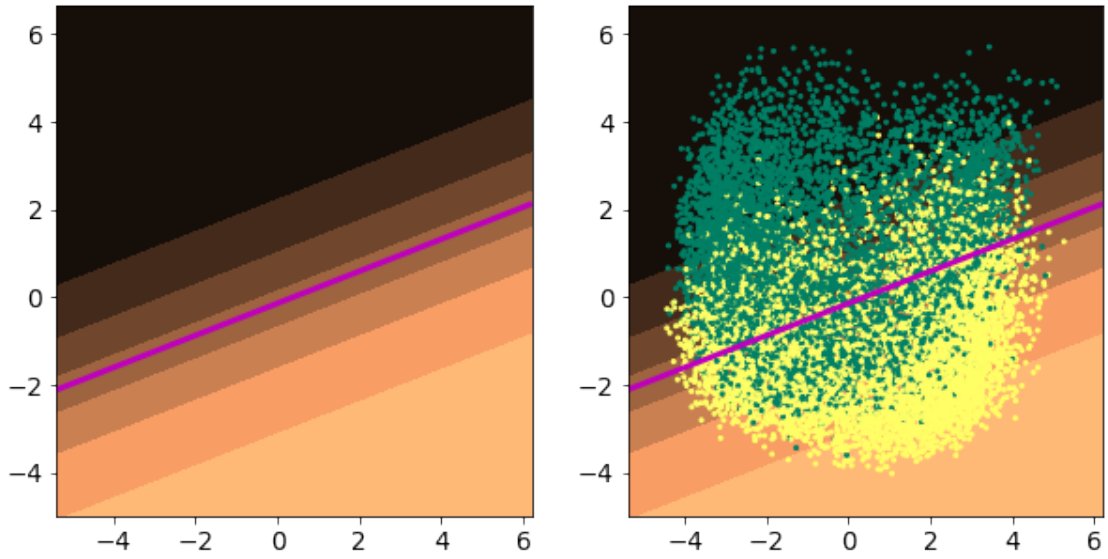
```
Error rate of Linear Logistic Regression Classifier (%): 26.610919822921787
NLL of Linear Logistic Regression Classifier: 1054.1279337524088
```

The next cell visualizes the classification border and the probabilistic map of the selected classifier (`clf`)

[10]:
```
def plot_proba_map(X, y, clf):
    """
    Plots a probabilistic map for classifier clf, and the coloured scatter plot␣
 ↪of the input samples

    Parameters
    ----------
    X:   Input Data Matrix (Size (N,2))
    y:   Targets (Size (N,))
    clf: A classifier object. It should have a predict_proba method
    """

    if X.shape[1] != 2:
        print('Can only plot 2D probability maps')
    else:
        # Create a regtangular grid.
        x_min, x_max = X[:, 0].min(), X[:, 0].max()
        y_min, y_max = X[:, 1].min(), X[:, 1].max()
        dx = x_max - x_min
```

```
        dy = y_max - y_min
        h = dy /400
        xx, yy = np.meshgrid(np.arange(x_min - 0.1 * dx, x_max + 0.1 * dx, h),
                             np.arange(y_min - 0.1 * dx, y_max + 0.1 * dy, h))
        X_grid = np.array([xx.ravel(), yy.ravel()]).T

        # Compute the classifier output for all samples in the grid.
        pp = clf.predict_proba(X_grid)[:,1]
        pp = pp.reshape(xx.shape)

        plt.figure(figsize=(10,5))
        plt.subplot(1, 2, 1)
        CS = plt.contourf(xx, yy, pp, cmap=plt.cm.copper)
        plt.contour(CS, levels=[0.5], colors='m', linewidths=(3,))
        plt.subplot(1, 2, 2)
        CS = plt.contourf(xx, yy, pp, cmap=plt.cm.copper)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=4, cmap='summer')
        plt.contour(CS, levels=[0.5], colors='m', linewidths=(3,))
        plt.show()
```

[11]:
```
plot_proba_map(X_tr_2D, y_tr_bin, clf)
```



A relevant parameter for the training of logistic regression is the optimization method used to update the parameter vector **w**. You can try other methods different from the `lbfgs` method which is the default selection. You can find information about these methods online, for instance in this stackoverflow entry: Logistic regression python solvers' definitions

2.1.3. Polynomial Logistic Regression

A strategy for the implementation of classifiers that provide non-linear boundaries consists of transforming the input variables. To this end, in this section we will use polynomial transformations, exploring different values of the degree of the built-in terms to try to optimize performance.

The proposed processing scheme for this section consists of the following steps:

- Polynomial expansion of the input variables using the `PolynomialFeatures` method of scikit-learn
- Scaling of all variables to zero mean and unit variance (`StandardScaler` method)
- Logistic regression

The free parameters to be adjusted are, therefore, the degree of the polynomial transformation and the parameter $ C $ of the logistic regression.

Exercise 2.4: Validate parameters `degree` and `C` of the proposed classification scheme using a 5-fold validation. Allow for polynomial transformations of degree up to 6. A very convenient way to do this is to define a processing pipeline in scikit-learn (using the `Pipeline` class), and use it together with `GridSearchCV`.

Note that both parameters should be validated together, i.e., all possible combinations need to be evaluated.

For the next two cells to work, you need to use the following variable names:

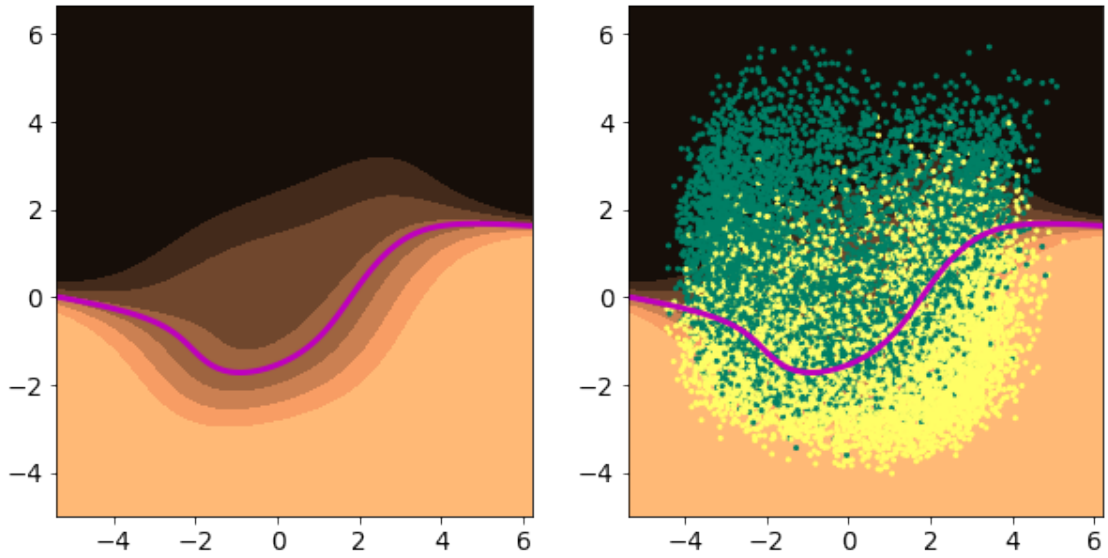
- `clf`: Best classifier selected with 5 fold strategy. It needs to implement a `clf.predict_proba` method
- `CE_poly`: Average Classification Error Rate of previous classifier calculated over the test data
- `NLL_poly`: Negative log-lilelihood of previous classifier calculated over the test data

[14]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler

# Write your code here
# <SOL>
parameters = {'lr__C': [10000, 1000, 100, 10, 1, 0.1, 0.01, 0.001, 0.0001],
              'poly__degree': [1, 2, 3, 4, 5, 6]}

pipe = Pipeline([('poly', PolynomialFeatures()),
                 ('scaler', StandardScaler()),
                 ('lr', LogisticRegression(max_iter=500, tol=1e-3))])

clf = GridSearchCV(pipe, parameters, cv=5)
clf.fit(X_tr_2D, y_tr_bin)

CE_poly  = 100 - 100*np.mean(y_tst_bin==clf.predict(X_tst_2D))
q_pred = clf.predict_log_proba(X_tst_2D)
NLL_poly = - np.sum(y_tst_bin.T @ q_pred[:,1]) - np.sum((1-y_tst_bin).T @␣
 ↪q_pred[:,0])
```

```
# </SOL>

print('Error rate of Polynomial Logistic Regression Classifier (%):', CE_poly)
print('NLL of Polynomial Logistic Regression Classifier:', NLL_poly)
print('Selected parameters:', clf.best_params_)
```

```
Error rate of Polynomial Logistic Regression Classifier (%): 25.627151992129853
NLL of Polynomial Logistic Regression Classifier: 983.2914422109839
Selected parameters: {'lr__C': 1, 'poly__degree': 5}
```

[15]:
```
plot_proba_map(X_tr_2D, y_tr_bin, clf)
```



2.1.4. Multi-Layer Perceptron Rather than using a predefined type of transformation, such as polynomial, we can recur to neural networks with non-linear activation functions to learn the most appropriate representation of the input data to solve the classification problem.

Exercise 2.5: In this section you will use the `MLPClassifier method` provided by scikit-learn for the implementation of feed-forward networks. You will only need to adjust the following parameters:

- `activation`: The activation function for the units of the intermediate layer. You can test `relu` and `tanh`.
- `max_iter`: Number of iterations of the optimization method. Try different values and make sure that the algorithm has completely converged.
- `hidden_layer_sizes`: Use just one hidden layer with 20 units
- `alpha`: The L2 regularization parameters.

Save your results using the following variable names:

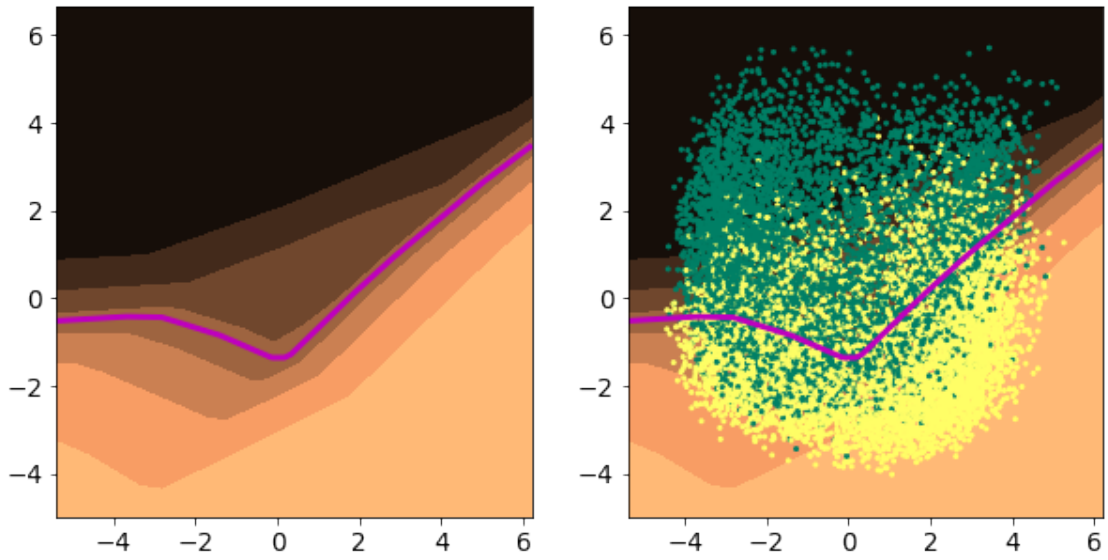
- `clf`: Best classifier selected with 5 fold strategy. It needs to implement a `clf.predict_proba` method

- CE_MLP: Average Classification Error Rate of previous classifier calculated over the test data
- NLL_MLP: Negative log-lilelihood of previous classifier calculated over the test data

[16]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import MLPClassifier

# <SOL>
parameters = {'activation':('relu', 'tanh'), 'max_iter':[2000, 4000],
              'hidden_layer_sizes': ((20,)), 'alpha': [100, 1, 0.01, 0.0001]}
clf = GridSearchCV(MLPClassifier(), parameters, cv=5)
clf.fit(X_tr_2D, y_tr_bin)

CE_MLP  = 100 - 100*np.mean(y_tst_bin==clf.predict(X_tst_2D))
y_pred = clf.predict_log_proba(X_tst_2D)
NLL_MLP = - np.sum(y_tst_bin.T @ y_pred[:,1]) - np.sum((1-y_tst_bin).T @␣
 ↪y_pred[:,0])
# </SOL>

print('CE of MLP Classifier (%):', CE_MLP)
print('NLL of MLP Classifier:', NLL_MLP)
print('Selected parameters:', clf.best_params_)
```

```
CE of MLP Classifier (%): 26.364977865223807
NLL of MLP Classifier: 1009.3704226569928
Selected parameters: {'activation': 'relu', 'alpha': 1, 'hidden_layer_sizes':
20, 'max_iter': 2000}
```

[17]:
```
plot_proba_map(X_tr_2D, y_tr_bin, clf)
```



2.2. Classification with all input features

Exercise 2.6: Analyze the performance in the binary classification problem of the following classifiers:

- Logistic regression (validate C)
- K-nearest neighbors (validate K)
- Multi-layer perceptron (validate size of hidden layers)

For comparison purposes you can use both the average classifiation error rate and the negative log-likelihood.

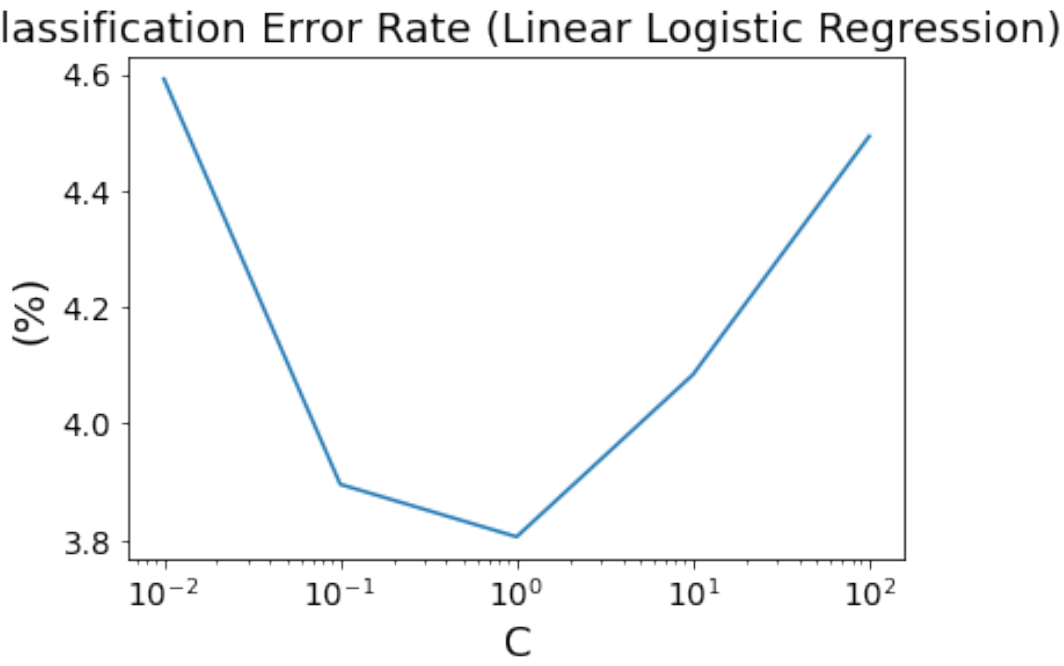
In this section you need to use all 784 available features.

2.2.1. Logistic Regression

[18]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# <SOL>
C_param = [100, 10, 1, 0.1, 0.01]
parameters = {'C': C_param}

clf = GridSearchCV(LogisticRegression(max_iter=2500, tol=1e-3), parameters,␣
 ↪cv=5)
clf.fit(X_tr_bin, y_tr_bin)

CE_param = 100 - 100 * clf.cv_results_['mean_test_score']
CE_LR  = 100 - 100*np.mean(y_tst_bin==clf.predict(X_tst_bin))
y_pred = clf.predict_log_proba(X_tst_bin)
NLL_LR = - np.sum(y_tst_bin.T @ y_pred[:,1]) - np.sum((1-y_tst_bin).T @ y_pred[:
 ↪,0])
# </SOL>

plt.semilogx(C_param, CE_param)
plt.xlabel('C'), plt.ylabel('(%)'), plt.title('Classification Error Rate␣
 ↪(Linear Logistic Regression)')
plt.show()

print('CE of Linear Logistic Regression Classifier (%):', CE_LR)
print('NLL of Linear Logistic Regression Classifier:', NLL_LR)
```

```
/opt/anaconda3/lib/python3.8/site-
packages/sklearn/linear_model/_logistic.py:814: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
```

```
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```




```
CE of Linear Logistic Regression Classifier (%): 4.8204623708804775
NLL of Linear Logistic Regression Classifier: 295.47427868944476
```

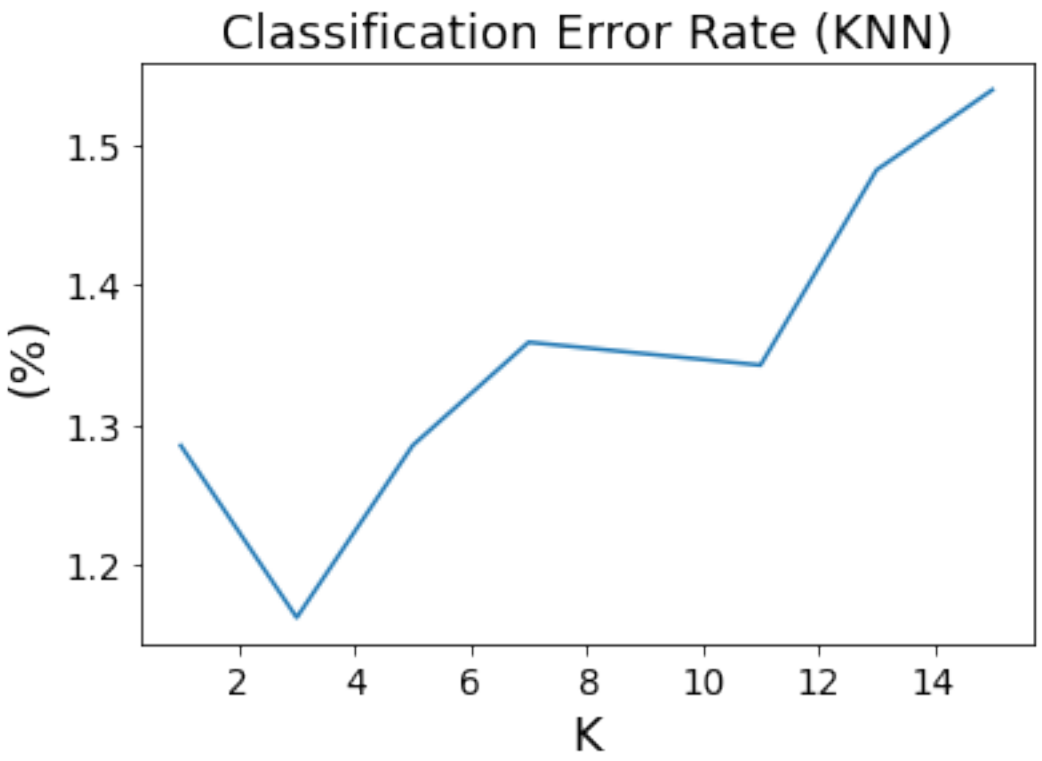
2.2.2. K Nearest Neighbors Be aware that, in this case, the number of test samples together with the dimension of the input data implies long processing times for classification (close to 1 min per execution in Google Colab). You might consider to subsample the test partition for the validation process.

[19]:
```python
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier

# <SOL>
K_param = [1, 3, 5, 7, 9, 11, 13, 15]
parameters = {'n_neighbors': K_param}

clf = GridSearchCV(KNeighborsClassifier(), parameters, cv=5, verbose=2)
clf.fit(X_tr_bin, y_tr_bin)

CE_param = 100 - 100 * clf.cv_results_['mean_test_score']
CE_KNN  = 100 - 100*np.mean(y_tst_bin==clf.predict(X_tst_bin))
# </SOL>
```

```
plt.plot(K_param, CE_param)
plt.xlabel('K'), plt.ylabel('(%)'), plt.title('Classification Error Rate (KNN)')
plt.show()

print('CE of KNN (%):', CE_KNN)
```

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits
[CV] END …n_neighbors=1; total time=   0.4s
[CV] END …n_neighbors=1; total time=   0.3s
[CV] END …n_neighbors=1; total time=   0.3s
[CV] END …n_neighbors=1; total time=   0.3s
[CV] END …n_neighbors=1; total time=   0.4s
[CV] END …n_neighbors=3; total time=   0.4s
[CV] END …n_neighbors=3; total time=   0.4s
[CV] END …n_neighbors=3; total time=   0.4s
[CV] END …n_neighbors=3; total time=   0.4s
[CV] END …n_neighbors=3; total time=   0.4s
[CV] END …n_neighbors=5; total time=   0.5s
[CV] END …n_neighbors=5; total time=   0.5s
[CV] END …n_neighbors=5; total time=   0.5s
[CV] END …n_neighbors=5; total time=   0.5s
[CV] END …n_neighbors=5; total time=   0.5s
[CV] END …n_neighbors=7; total time=   0.5s
[CV] END …n_neighbors=7; total time=   0.5s
[CV] END …n_neighbors=7; total time=   0.5s
[CV] END …n_neighbors=7; total time=   0.5s
[CV] END …n_neighbors=7; total time=   0.5s
[CV] END …n_neighbors=9; total time=   0.5s
[CV] END …n_neighbors=9; total time=   0.5s
[CV] END …n_neighbors=9; total time=   0.5s
[CV] END …n_neighbors=9; total time=   0.5s
[CV] END …n_neighbors=9; total time=   0.5s
[CV] END …n_neighbors=11; total time=   0.5s
[CV] END …n_neighbors=11; total time=   0.5s
[CV] END …n_neighbors=11; total time=   0.5s
[CV] END …n_neighbors=11; total time=   0.5s
[CV] END …n_neighbors=11; total time=   0.5s
[CV] END …n_neighbors=13; total time=   0.5s
[CV] END …n_neighbors=13; total time=   0.5s
[CV] END …n_neighbors=13; total time=   0.5s
[CV] END …n_neighbors=13; total time=   0.5s
[CV] END …n_neighbors=13; total time=   0.5s
[CV] END …n_neighbors=15; total time=   0.5s
[CV] END …n_neighbors=15; total time=   0.5s
[CV] END …n_neighbors=15; total time=   0.5s
[CV] END …n_neighbors=15; total time=   0.5s
```

```
[CV] END …n_neighbors=15; total time=   0.5s
```




```
CE of KNN (%): 1.0329562223315207
```

2.2.3. Multi-Layer Perceptron

Be aware that, in this case, the number of test samples together with the dimension of the input data implies long processing times for classification (close to 1 min per execution in Google Colab). You might consider to subsample the test partition for the validation process.

[ ]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import MLPClassifier

# <SOL>
parameters = {'activation':('relu', 'tanh'), 'max_iter':[2000, 4000],
              'hidden_layer_sizes': ((400,100), (400,200), (200,100))}
clf = GridSearchCV(MLPClassifier(), parameters, cv=5, verbose=2)
clf.fit(X_tr_bin, y_tr_bin)

CE_MLP  = 100 - 100*np.mean(y_tst_bin==clf.predict(X_tst_bin))
y_pred = clf.predict_proba(X_tst_bin)
y_pred = np.log(y_pred + 1e-6)
NLL_MLP = - np.sum(y_tst_bin.T @ y_pred[:,1]) - np.sum((1-y_tst_bin).T @␣
 ↪y_pred[:,0])
```

```
# </SOL>

print('CE of MLP Classifier (%):', CE_MLP)
print('NLL of MLP Classifier:', NLL_MLP)
print('Selected parameters:', clf.best_params_)
```

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits
[CV] END activation=relu, hidden_layer_sizes=(400, 100), max_iter=2000; total
time=  14.0s
[CV] END activation=relu, hidden_layer_sizes=(400, 100), max_iter=2000; total
time=  13.7s
[CV] END activation=relu, hidden_layer_sizes=(400, 100), max_iter=2000; total
time=  15.3s
[CV] END activation=relu, hidden_layer_sizes=(400, 100), max_iter=2000; total
time=  16.4s
[CV] END activation=relu, hidden_layer_sizes=(400, 100), max_iter=2000; total
time=  20.5s
[CV] END activation=relu, hidden_layer_sizes=(400, 100), max_iter=4000; total
time=  17.8s
[CV] END activation=relu, hidden_layer_sizes=(400, 100), max_iter=4000; total
time=  16.4s
[CV] END activation=relu, hidden_layer_sizes=(400, 100), max_iter=4000; total
time=  18.3s
[CV] END activation=relu, hidden_layer_sizes=(400, 100), max_iter=4000; total
time=  18.9s
[CV] END activation=relu, hidden_layer_sizes=(400, 100), max_iter=4000; total
time=  19.3s
[CV] END activation=relu, hidden_layer_sizes=(400, 200), max_iter=2000; total
time=  17.7s
[CV] END activation=relu, hidden_layer_sizes=(400, 200), max_iter=2000; total
time=  17.4s
[CV] END activation=relu, hidden_layer_sizes=(400, 200), max_iter=2000; total
time=  16.4s
[CV] END activation=relu, hidden_layer_sizes=(400, 200), max_iter=2000; total
time=  17.8s
[CV] END activation=relu, hidden_layer_sizes=(400, 200), max_iter=2000; total
time=  20.3s
[CV] END activation=relu, hidden_layer_sizes=(400, 200), max_iter=4000; total
time=  16.4s
[CV] END activation=relu, hidden_layer_sizes=(400, 200), max_iter=4000; total
time=  18.5s
[CV] END activation=relu, hidden_layer_sizes=(400, 200), max_iter=4000; total
time=  16.3s
[CV] END activation=relu, hidden_layer_sizes=(400, 200), max_iter=4000; total
time=  19.7s
[CV] END activation=relu, hidden_layer_sizes=(400, 200), max_iter=4000; total
time=  18.1s
```

```
[CV] END activation=relu, hidden_layer_sizes=(200, 100), max_iter=2000; total
time=  10.1s
[CV] END activation=relu, hidden_layer_sizes=(200, 100), max_iter=2000; total
time=   9.1s
[CV] END activation=relu, hidden_layer_sizes=(200, 100), max_iter=2000; total
time=  11.2s
[CV] END activation=relu, hidden_layer_sizes=(200, 100), max_iter=2000; total
time=  10.2s
[CV] END activation=relu, hidden_layer_sizes=(200, 100), max_iter=2000; total
time=   9.0s
[CV] END activation=relu, hidden_layer_sizes=(200, 100), max_iter=4000; total
time=  10.3s
[CV] END activation=relu, hidden_layer_sizes=(200, 100), max_iter=4000; total
time=   9.4s
[CV] END activation=relu, hidden_layer_sizes=(200, 100), max_iter=4000; total
time=  10.6s
[CV] END activation=relu, hidden_layer_sizes=(200, 100), max_iter=4000; total
time=  10.2s
[CV] END activation=relu, hidden_layer_sizes=(200, 100), max_iter=4000; total
time=  11.5s
[CV] END activation=tanh, hidden_layer_sizes=(400, 100), max_iter=2000; total
time=  24.9s
[CV] END activation=tanh, hidden_layer_sizes=(400, 100), max_iter=2000; total
time=  24.6s
[CV] END activation=tanh, hidden_layer_sizes=(400, 100), max_iter=2000; total
time=  24.6s
```

2.3 3. Multi Class Classification

In this section we will train classification schemes that allow us to discriminate between the 10 digits that make up the complete dataset. In this case, you must use the parameters provided, and you will be asked to study the execution times required by the different methods. To do this, you can use the `time` library as follows:

```
import time

start = time.time()
#Some code should go here
etime = time.time() - start
```

2.3.1 3.1. Principal Component Analysis (PCA)

In a preliminary way, and as you did in section 2.2.1, we can analyze the first two PCA components of the available data. You can see that in this case there is a very important overlap between the digits of all the classes on the first two components of PCA.

In this section, we will apply the different classification strategies **using all the available features**.

Exercise 3.1: Analyze the variance of the successive projections that the PCA method would obtain, and reflect on whether a smaller-dimensional representation of the input data could be

used.

[ ]:
```
from sklearn.decomposition import PCA

pca = PCA(2)  # project from 784 to 2 dimensions
projected = pca.fit_transform(X_tr)
plt.scatter(projected[:, 0], projected[:, 1],
            c=y_tr, edgecolor='none', alpha=0.5,
            cmap=plt.cm.get_cmap('rainbow', 10))
plt.xlabel('component 1')
plt.ylabel('component 2')
plt.colorbar();
```

[ ]:
```
# Solution to Exercise 3.1

# <SOL>
pca = PCA(100)  # project from 784 to 100 dimensions
projected = pca.fit_transform(X_tr)
plt.plot(pca.explained_variance_)
plt.xlabel('Projection')
plt.ylabel('Explained variance')
plt.show()
# </SOL>
```

2.3.2 3.2. Nearest Neighbor Method

In this section, you will analyze the performance of the nearest neighbor (1-NN) algorithm. The complexity of this algorithm grows with the size of the training set, so it is proposed to analyze the behavior of the algorithm for a variable size of the training set.

Exercise 3.2: Use the 1-NN method with a varying training set size. Obtain for each size:

- The average classifiation error rate calculated on the test set
- The fit time for the 1-NN method
- The time taken to classify the complete test partition (10000 samples)

[ ]:
```
from sklearn.neighbors import KNeighborsClassifier
import time

train_size = [250, 500, 1000, 2000, 5000, 10000, 25000, 60000]
fit_time = []
test_time = []
CE = []

for ntrain in train_size:
    print('Classifying with', ntrain, 'samples')
    # Write your code here
    # <SOL>
    start = time.time()
```

```
    knn = KNeighborsClassifier()
    knn.fit(X_tr[:ntrain,:], y_tr[:ntrain])
    etime = time.time() - start
    fit_time.append(etime)
    start = time.time()
    CE.append(np.mean(knn.predict(X_tst)!=y_tst))
    etime = time.time() - start
    test_time.append(etime)
    # </SOL>
```

[ ]:
```
plt.figure(figsize=(15,3.5))
plt.subplot(1, 3, 1), plt.plot(train_size, 100*np.array(CE)), plt.
 ↪xlabel('Training Set Size'), plt.ylabel('(%)'), plt.title('CE')
plt.subplot(1, 3, 2), plt.plot(train_size, fit_time), plt.xlabel('Training Set␣
 ↪Size'), plt.ylabel('Seconds'), plt.title('Fit Time')
plt.subplot(1, 3, 3), plt.plot(train_size, test_time), plt.xlabel('Training Set␣
 ↪Size'), plt.ylabel('Seconds'), plt.title('Test Time')
plt.show()
print('Average Classification Error for the 1-NN approach', 100*np.min(CE))
```

Exercise 3.3: Calculate the confusion matrix of the 1-NN classifier when using 1000 samples for the training set, and answer the following questions: - Which two digits are most frequently confused? - Which is the digit that gets misclassified most often?

[ ]:
```
# <SOL>
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

ntrain = 1000
knn = KNeighborsClassifier()
knn.fit(X_tr[:ntrain,:], y_tr[:ntrain])

cm = confusion_matrix(y_tst, knn.predict(X_tst), labels=knn.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=knn.classes_)
clg = disp.plot()
clg.figure_.set_size_inches(7,7)
plt.show()
# </SOL>
```

2.3.3 3.3. Multi-Layer Perceptron

Exercise 3.4: Train an MLP network using all samples in the training set. Use the following settings: - `relu` activation function for the hidden units - Two hidden layers with 200 and 100 neurons, respectively - Maximum number of iterations for the training: 2000 iterations

Compare the performance of the MLP network with that of the 1-NN method. Consider in your comparison both the classification error rate and the fit and operation times.

Calculate also the negative log likelihood of the model implemented by the Neural Network.

[ ]:
```
from sklearn.neural_network import MLPClassifier

MLP = MLPClassifier(activation='relu', max_iter=2000,␣
 ↪hidden_layer_sizes=(200,100))

# <SOL>
start = time.time()
MLP.fit(X_tr, y_tr)
print('Fit time for the MLP classifier:', time.time() - start, 'seconds')
start = time.time()
CE = 100 - 100 * np.mean(y_tst==MLP.predict(X_tst))
print('Test time for the MLP classifier (10000 samples):', time.time() - start,␣
 ↪'seconds')
print('Classification error rate for the MLP classifer:', CE, '%')

# The following two lines are equivalent to the calculation below
# They only differ in which log_loss normalizes over the number of samples
# from sklearn.metrics import log_loss
# log_loss(y_tst, MLP.predict_proba(X_tst))
y_pred = MLP.predict_proba(X_tst)
y_pred = np.log(y_pred + 1e-15)
NLL_MLP = 0
for k in np.arange(10):
    NLL_MLP -= np.sum(y_pred[y_tst==k, k])

# </SOL>
print('Negative Log Likelihood for the MLP classifier:', NLL_MLP)
```

3 Part 2. Implementing Deep Networks with PyTorch

3.1 4. Pytorch Tutorial

- Pytorch is a Python library that provides different levels of abstraction for implementing deep neural networks
- The main features of PyTorch are:
  - Definition of numpy-like n-dimensional *tensors*. They can be stored in / moved to GPU for parallel execution of operations
  - Automatic calculation of gradients, making *backward gradient calculation* transparent to the user
  - Definition of common loss functions, NN layers of different types, optimization methods, data loaders, etc, simplifying NN implementation and training
  - Provides different levels of abstraction, thus a good balance between flexibility and simplicity
- This notebook provides just a basic review of the main concepts necessary to train NNs with PyTorch taking materials from:

  - Learning PyTorch with Examples, by Justin Johnson
  - What is *torch.nn* really?, by Jeremy Howard
  - Pytorch Tutorial for Deep Learning Lovers, by Kaggle user kanncaa1

3.1.1 4.1. PyTorch Installation

- PyTorch can be installed with or without GPU support
  - If you have an Anaconda installation, you can install from the command line, using the instructions of the project website
- PyTorch is also preinstalled in Google Collab with free GPU access
  - Follow RunTime -> Change runtime type, and select GPU for HW acceleration
- Please, refer to Pytorch getting started tutorial for a quick introduction regarding tensor definition, GPU vs CPU storage of tensors, operations, and bridge to Numpy

3.1.2 4.2. Torch tensors (very) general overview

Esentially, tensors are objects provided by PyTorch for numerical representation. They are generic n-dimensional arrays such as the ones used by Numpy. Apart from the library providing them, there are two important differences between Numpy arrays and PyTorch tensors:

- Tensors can be stored in / moved to GPU. When doing so, certain operations will be parallelized resulting in faster execution.
- Tensors provide out-of-the-box functions for tracking the operations in which they are involved, and to systematically compute derivatives, something which is very useful for the implementation of the backpropagation method used for training deep networks.

Creating tensors from Numpy arrays

We can create tensors with different construction methods provided by the library, either to create new tensors from scratch or from a Numpy array

Tensors can be converted back to numpy arrays

Note that in this case, a tensor and its corresponding numpy array **will share memory**

[ ]:
```
import torch

x = torch.rand((100,200))
X_tr_tensor = torch.from_numpy(X_tr)

print(x.type())
print(X_tr_tensor.size())
```

Operations and slicing with tensors

Operations and slicing involving tensors use a syntax similar to that of numpy

[ ]:
```
print('Size of tensor x:', x.size())
print('Tranpose of vector has size', x.t().size()) #Transpose and compute size
print('Extracting upper left matrix of size 3 x 3:', x[:3,:3])
print((x @ x.t()).size())
xpx = x.add(x)
```

```
xpx2 = torch.add(x,x)
print((xpx!=xpx2).sum())   #Since all are equal, count of different terms is␣
 ↪zero
```

Adding underscore performs operations "*in place*", e.g., `x.add_(y)`

Parallelization of Operations using GPUs

- If a GPU is available, tensors can be moved to and from the GPU device
- Operations on tensors stored in a GPU will be carried out using GPU resources and will typically be highly parallelized

[ ]:
```
if torch.cuda.is_available():
    device = torch.device('cuda')
    x = x.to(device)
    y = x.add(x)
    y = y.to('cpu')
else:
    print('No GPU card is available')
```

Note: If you are using Google Colab and the previous cell indicates that you do not have access to a GPU, you may change your runtime type. However, note that doing so will restart your runtime, so that you will have to run again the initial cells of the notebook to load the data.

3.1.3 4.3. Automatic Gradient Calculation

- PyTorch tensors have a property `requires_grad`. When true, PyTorch automatic gradient calculation will be activated for that variable
- In order to compute these derivatives numerically, PyTorch keeps track of all operations carried out on these variables, organizing them in a forward computation graph.
- When executing the `backward()` method, derivatives will be calculated
- However, this should only be activated when necessary, to save computation

[ ]:
```
x.requires_grad = True
y = (3 * torch.log(x)).sum()
y.backward()
print(x.grad[:2,:2])
print(3/x[:2,:2])

x.requires_grad = False
x.grad.zero_()
print('Automatic gradient calculation is deactivated, and gradients set to␣
 ↪zero')
```

Exercise 4.1:

- Initialize a tensor `x` as `X_tr[0,315:325]`
- Compute output vector `y` applying a function of your choice to `x`

- Compute scalar value `z` as the sum of all elements in `y` squared
- Check that `x.grad` calculation is correct using the `backward` method
- Try to run your cell multiple times to see if the calculation is still correct. If not, implement the necessary modifications so that you can run the cell multiple times, but the gradient does not change from run to run

Note: The backward method can only be run on scalar variables

[ ]:
```
if torch.cuda.is_available():
    x = torch.from_numpy(X_tr[0,315:325]).to(device)
else:
    x = torch.from_numpy(X_tr[0,315:325])
```

[ ]:
```
# <SOL>
x.requires_grad = True
y = x.add(torch.exp(x))
z = (y**2).sum()
z.backward()
print(x.grad)
print(2*y*(1 + torch.exp(x)))

x.requires_grad = False
x.grad.zero_()
print('Automatic gradient calculation is deactivated, and gradients set to␣
 ↪zero')
# </SOL>
```

3.2 5. Feed Forward Networks using PyTorch

In this section we are going to illustrate how we can implement a multilayer perceptron-type neural network using the features of PyTorch. The network thus implemented will be equivalent to the one you would train using the scikit-learn `MLP` function, but thanks to the use of PyTorch it will be able to be executed parallelizing many of the calculations in GPU, which will allow a much faster training.

A first possibility would be the direct implementation of the neural network, through an implementation based on PyTorch tensors of the evaluation functions of the neural network and of the derivatives of the cost function with respect to the different parameters (back-propagation). However, PyTorch module `nn` provides different levels of abstraction that considerably simplify the implementation of the network, in addition to making the theoretical calculation of derivatives unnecessary.

Before proceding, we need to import training and test data as PyTorch tensors. The fragment below can be used for that purpose. Note that if you are using a GPU, all tensors should be moved to the GPU.

Exercise 5.1: Complete the code below to create PyTorch tensors for the different MNIST variables. The tensors for the input data have already been created for you. For encoding class

membership, you need to create `y_tr` and `y_val` using One-Hot-Encoding. You can easily do that using sklearn method `LabelBinarizer`.

[ ]:
```
from sklearn.preprocessing import LabelBinarizer

#Convert to Torch tensors. Float type is required
X_tr_torch = torch.from_numpy(X_tr).float()
X_val_torch = torch.from_numpy(X_tst).float()
# <SOL>
#Note we are using One Hot Encoding for the labels
lb = LabelBinarizer()
y_tr_torch = torch.from_numpy(lb.fit_transform(y_tr))
y_val_torch = torch.from_numpy(lb.transform(y_tst))
# </SOL>
```

3.2.1 5.1. Using torch `nn.Module` and `nn.Parameter`

PyTorch `nn` module provides many attributes and methods to make simple the implementation and training of Neural Networks.

`nn.Module` and `nn.Parameter` allow to implement a concise network configuration, and simplify the calculation of the gradients

- `nn.Module` is a PyTorch class that will be used to encapsulate and design a specific neural network, thus, it is central to the implementation of deep neural nets using PyTorch
- `nn.Parameter` allow the definition of trainable network parameters. In this way, we will simplify the implementation of the training loop.
- All parameters defined with `nn.Parameter` will have `requires_grad = True`

Below you can see a PyTorch fragment for the definition of a single layer perceptron (SLP) network. You can see that at least two methods need to be defined: the initialization of the network (including parameter definition and initialization), and a `forward` method that implements how the network produces its output for a given input pattern. Other auxiliary functions may be defined as well.

However, you can see that there is no need to implement a `backward` method for gradient calculation.

```
from torch import nn

class my_multiclass_net(nn.Module):
    def __init__(self, nin, nout):
        """This method initializes the network parameters
        Parameters nin and nout stand for the number of input parameters (features in X)
        and output parameters (number of classes)"""
        super().__init__()
        self.W = nn.Parameter(.1 * torch.randn(nin, nout))
        self.b = nn.Parameter(torch.zeros(nout))

    def forward(self, x):
```

```
        return softmax(x @ self.W + self.b)

    def softmax(t):
        """Compute softmax values for each sets of scores in t"""
        return t.exp() / t.exp().sum(-1).unsqueeze(-1)
```

You can see that by using `nn.Parameter` and `nn.Module` you can easily implement any function of your choice. However, we need to be careful about matrix dimensions and some particularities which are required to correctly operate PyTorch tensors.

For standard feed-forward networks such as MLPs, we can use other PyTorch abstraction levels that make these implementations even simpler.

- `nn.Module` comes with several kinds of pre-defined layers, thus making it even simpler to implement neural networks
- `nn.CrossEntropyLoss` implements the calculation of the negative log likelihood incorporating the softmax for the predictions (so there is no need to include it in the `forward` method of the network

The code below shows how these predefined layers and cost functions can be used to create an SLP network in a rather straightforward manner. Note that when creating the network we just need to specify the dimensionality of the input and output data, i.e., number of input features and number of classes.

[ ]:
```
from torch import nn

class my_multiclass_net(nn.Module):
    def __init__(self, nin, nout):
        """Note that now, we do not even need to initialize network parameters␣
 ↪ourselves"""
        super().__init__()
        self.lin = nn.Linear(nin, nout)

    def forward(self, x):
        return self.lin(x)

loss_func = nn.CrossEntropyLoss()
```

The code below implements the training of the network using conventional gradient descent. The training takes place over a predefined number of epochs using a fixed step size.

It is important to note that:

- Gradient updates are stopped after the evaluation of the network output for all training patterns. This is done by encapsulating any additional computations inside a block `with torch.no_grad()`.
- Parameter updates are implemented by iterating over all network parameter using the method `nn.Model.parameters()`
- After parameter update, gradients are set back to zero for the next epoch

[ ]:
```
def CE(y, y_hat):
    return 1-(y.argmax(axis=-1) == y_hat.argmax(axis=-1)).float().mean()

my_net = my_multiclass_net(X_tr_torch.size()[1], y_tr_torch.size()[1])
epochs = 300
rho = .1

loss_tr = np.zeros(epochs)
loss_val = np.zeros(epochs)
CE_tr = np.zeros(epochs)
CE_val = np.zeros(epochs)

start = time.time()

for epoch in range(epochs):

    print(f'Current epoch: {epoch+1}    \r', end="")

    #Compute network output and cross-entropy loss
    pred = my_net(X_tr_torch)
    loss = loss_func(pred, y_tr_torch.argmax(axis=-1))

    #Compute gradients
    loss.backward()

    #Deactivate gradient automatic updates
    with torch.no_grad():
        #Computing network performance after iteration
        loss_tr[epoch] = loss.item()
        CE_tr[epoch] = CE(y_tr_torch, pred).item()
        pred_val = my_net(X_val_torch)
        loss_val[epoch] = loss_func(pred_val, y_val_torch.argmax(axis=-1)).
 ↪item()
        CE_val[epoch] = CE(y_val_torch, pred_val).item()

        #Weight update
        for p in my_net.parameters():
            p -= p.grad * rho
        #Reset gradients
        my_net.zero_grad()

print('Training of the network took', time.time()-start, 'seconds')
```

[ ]:
```
plt.figure(figsize=(14,5))
plt.subplot(1, 2, 1), plt.plot(loss_tr, 'b'), plt.plot(loss_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Cross-entropy loss')
```

```
plt.subplot(1, 2, 2), plt.plot(CE_tr, 'b'), plt.plot(CE_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Classification Error')
plt.show()
```

5.2. Network Optimization

We cover in this subsection two different aspects about network training using PyTorch:

- Using `torch.optim` allows an easier and more interpretable encoding of neural network training, and opens the door to more sophisticated training algorithms
- Using **minibatches** can speed up network convergence. The idea of minibatches is that, at each epoch, gradients are evaluated over just a subset of the training input data. Training of the network will normally require more epochs but, as each epoch requires the evaluation of the network output for a smaller subset of training samples, the overall training time is normally reduced significantly.

`torch.optim` provides two convenient methods for neural network training: * `opt.step()` updates all network parameters using current gradients * `opt.zero_grad()` resets all network parameters

[ ]:
```
from torch.utils.data import TensorDataset, DataLoader

train_ds = TensorDataset(X_tr_torch, y_tr_torch)
train_dl = DataLoader(train_ds, batch_size=64)
```

[ ]:
```
from torch import optim

my_net = my_multiclass_net(X_tr_torch.size()[1], y_tr_torch.size()[1])
opt = optim.SGD(my_net.parameters(), lr=0.1)

epochs = 150

loss_tr = np.zeros(epochs)
loss_val = np.zeros(epochs)
CE_tr = np.zeros(epochs)
CE_val = np.zeros(epochs)

start = time.time()
for epoch in range(epochs):

    print(f'Current epoch: {epoch+1}    \r', end="")

    # In each epoch we iterate over all minibatches
    for xb, yb in train_dl:

        #Compute network output and cross-entropy loss for current minibatch
        pred = my_net(xb)
        loss = loss_func(pred, yb.argmax(axis=-1))
```

```
        #Compute gradients and optimize parameters
        loss.backward()
        opt.step()
        opt.zero_grad()

    #At the end of each epoch, evaluate overall network performance
    with torch.no_grad():
        #Computing network performance after iteration
        pred = my_net(X_tr_torch)
        loss_tr[epoch] = loss_func(pred, y_tr_torch.argmax(axis=-1)).item()
        CE_tr[epoch] = CE(y_tr_torch, pred).item()
        pred_val = my_net(X_val_torch)
        loss_val[epoch] = loss_func(pred_val, y_val_torch.argmax(axis=-1)).
 ↪item()
        CE_val[epoch] = CE(y_val_torch, pred_val).item()

print('Neural Network training completed in', time.time()-start, 'seconds')
```

[ ]:
```
plt.figure(figsize=(14,5))
plt.subplot(1, 2, 1), plt.plot(loss_tr, 'b'), plt.plot(loss_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(100*CE_tr, 'b'), plt.plot(100*CE_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Classification Error (%)')
plt.show()
```

Comparing this figures to those for the conventional gradient descent method, we can extract a number of conclusions:

- Convergence is radically faster when using SGD with minibatches. Note that just after the first epoch the error (both in terms of loss function and CE) is already smaller than that achieved with conventional gradient descent
- In this case, the error in the validation set starts increasing slightly after a number of epochs. I.e., even for a linear classifier overfitting may occur given the high dimensionality of the input data
- Note that the final classification error is much larger than that observed in Section 3; however keep in mind that the network we have just implemented is constrained to linear classification.

Exercise 5.2: Implement network training with other optimization methods. You can refer to the official documentation and select a couple of methods. You can also try to implement adaptive learning rates using `torch.optim.lr_scheduler`

5.3. Multi Layer networks using `nn.Sequential`

As we have seen, PyTorch simplifies considerably the implementation of neural network training, since we do not need to implement derivatives ourselves.

We can also make a simpler implementation of multilayer networks using `nn.Sequential` function. It returns directly a network with the requested topology, including parameters **and forward evaluation method**

For instance, the cell below defines a Network with two hidden layers with 200 and 100 units for the resolution of the MNIST multiclass problem. `Relu` activation is used at the output of the neurons of the hidden layers.

[ ]:
```
my_MLP_net = nn.Sequential(
    nn.Linear(X_tr_torch.size()[1], 200),
    nn.ReLU(),
    nn.Linear(200,100),
    nn.ReLU(),
    nn.Linear(100,y_tr_torch.size()[1])
)
```

Exercise 5.3: Train the MLP network we have just defined on the MNIST dataset using the following settings: * Loss function: `nn.CrossEntropyLoss()` * Optimization algorithm: `optim.Adam()` with learning rate `1e-4` * Minibatch size: 256 * Number of epochs: 100

Calculate the time required to train the network

[ ]:
```
# <SOL>
from torch import nn
from torch import optim
from torch.utils.data import TensorDataset, DataLoader
import time

train_ds = TensorDataset(X_tr_torch, y_tr_torch)
train_dl = DataLoader(train_ds, batch_size=256)

loss_func = nn.CrossEntropyLoss()
opt = optim.Adam(my_MLP_net.parameters(), lr=1e-4)

epochs = 100

loss_tr = np.zeros(epochs)
loss_val = np.zeros(epochs)
CE_tr = np.zeros(epochs)
CE_val = np.zeros(epochs)

start = time.time()
for epoch in range(epochs):

    print(f'Current epoch: {epoch+1}    \r', end="")

    #In each epoch we iterate over all minibatches
    for xb, yb in train_dl:

        #Compute network output and cross-entropy loss for current minibatch
        pred = my_MLP_net(xb)
        loss = loss_func(pred, yb.argmax(axis=-1))
```

```
        #Compute gradients and optimize parameters
        loss.backward()
        opt.step()
        opt.zero_grad()

    #At the **end of each epoch**, evaluate overall network performance
    with torch.no_grad():
        #Computing network performance after iteration
        pred = my_MLP_net(X_tr_torch)
        loss_tr[epoch] = loss_func(pred, y_tr_torch.argmax(axis=-1)).item()
        CE_tr[epoch] = CE(y_tr_torch, pred).item()
        pred_val = my_MLP_net(X_val_torch)
        loss_val[epoch] = loss_func(pred_val, y_val_torch.argmax(axis=-1)).
 ↪item()
        CE_val[epoch] = CE(y_val_torch, pred_val).item()

print('Neural Network training completed in', time.time()-start, 'seconds')

# </SOL>
```

[ ]:
```
plt.figure(figsize=(14,5))
plt.subplot(1, 2, 1), plt.plot(loss_tr, 'b'), plt.plot(loss_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(100*CE_tr, 'b'), plt.plot(100*CE_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Classification Error (%)')
plt.show()
```

Exercise 5.4: Modify your code so that network training is done using GPUs. Compare the training time when using CPU and GPU implementations.

[ ]:
```
# Network, training and validation data should be moved to GPU.
# You can do that with .to(device) or .cuda() functions

my_MLP_cuda = nn.Sequential(
    nn.Linear(X_tr_torch.size()[1], 200),
    nn.ReLU(),
    nn.Linear(200,100),
    nn.ReLU(),
    nn.Linear(100,y_tr_torch.size()[1])
).cuda()

X_tr_cuda = X_tr_torch.to(device)
X_val_cuda = X_val_torch.to(device)
y_tr_cuda = y_tr_torch.to(device)
y_val_cuda = y_val_torch.to(device)
```

```
# Adapt training code
# <SOL>
train_ds = TensorDataset(X_tr_cuda, y_tr_cuda)
train_dl = DataLoader(train_ds, batch_size=256)

loss_func = nn.CrossEntropyLoss()
opt = optim.Adam(my_MLP_cuda.parameters(), lr=1e-4)

epochs = 100

loss_tr = np.zeros(epochs)
loss_val = np.zeros(epochs)
CE_tr = np.zeros(epochs)
CE_val = np.zeros(epochs)

start = time.time()
for epoch in range(epochs):

    print(f'Current epoch: {epoch+1}    \r', end="")

    #In each epoch we iterate over all minibatches
    for xb, yb in train_dl:

        #Compute network output and cross-entropy loss for current minibatch
        pred = my_MLP_cuda(xb)
        loss = loss_func(pred, yb.argmax(axis=-1))

        #Compute gradients and optimize parameters
        loss.backward()
        opt.step()
        opt.zero_grad()

    #At the **end of each epoch**, evaluate overall network performance
    with torch.no_grad():
        #Computing network performance after iteration
        pred = my_MLP_cuda(X_tr_cuda)
        loss_tr[epoch] = loss_func(pred, y_tr_cuda.argmax(axis=-1)).item()
        CE_tr[epoch] = CE(y_tr_cuda, pred).item()
        pred_val = my_MLP_cuda(X_val_cuda)
        loss_val[epoch] = loss_func(pred_val, y_val_cuda.argmax(axis=-1)).item()
        CE_val[epoch] = CE(y_val_cuda, pred_val).item()

print('Neural Network training completed in', time.time()-start, 'seconds')

# </SOL>
```

[ ]:
```
plt.figure(figsize=(14,5))
plt.subplot(1, 2, 1), plt.plot(loss_tr, 'b'), plt.plot(loss_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(100*CE_tr, 'b'), plt.plot(100*CE_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Classification Error (%)')
plt.show()
```

Important: This is the end of material currently available for Neural Networks. For next years, we should extend in the following directions (at least): 1/ Generalización mediante distintas estrategias: otras funciones de coste, dropout, early stopping, términos de regularización, etc; 2/ La continuación natural para este notebook sería seguir con redes convolucionales, para lo cual podríamos seguir el tutorial que listo más abajo. Los alumnos podrían revisar también qué prestaciones han alcanzado en este dataset otras estrategias…

3.2.2 5.4. Generalization

- For complex network topologies (i.e., many parameters), network training can incur in overfitting issues
- Some common strategies to avoid this are:
  - Early stopping
  - Dropout regularization

Image Source

- Data augmentation can also be used to avoid overfitting, as well as to achieve improved accuracy by providing the network some a priori expert knowledge
  - E.g., if image rotations and scalings do not affect the correct class, we could enlarge the dataset by creating artificial images with these transformations

3.3 6. Convolutional Neural Networks

- Tutorial que podría sernos de utilidad
- Página web de MNIST en Wikipedia

[ ]:
```

```