RMLA_Etivity2_PaerhatiRemutula_16099958

August 4, 2021

Artificial Intelligence - MSc CS6501 - MACHINE LEARNING AND APPLICATIONS

Business Analytics - MSc ET5003 - MACHINE LEARNING APPLICATIONS

Annual Repeat

0.0.1 Instructor: Enrique Naredo

0.0.2 RepMLA_Etivity-2.0

[37]:
```
# Enter your details here:
Student_ID = "16099958" #@param {type:"string"}
Student_full_name = "Paerhati Remutula" #@param {type:"string"}
```

[39]:
```
#@title Current Date
Today = '2021-08-03' #@param {type:"date"}
```

[38]:
```
#@title Notebook information
Notebook_type = 'Etivity' #@param ["Example", "Lab", "Practice", "Etivity",␣
↪"Assignment", "Exam"]
Version = 'Final' #@param ["Draft", "Final"] {type:"raw"}
Submission = True #@param {type:"boolean"}
```

1 Introduction

Classification is the process of predicting the class of given data points.

- An easy to understand example is classifying emails as “spam” or “not spam.”
- In machine learning an algorithm learns how to assign a class label to examples from a problem domain.
- Classification belongs to the category of supervised learning where the targets also provided with the input data.

In this notebook we will solve a classification problem using the well-known Mnist dataset and the also well-known classifier algorithm Logistic Regression.

2 Dataset

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits.

- The MNIST database contains 60,000 training images and 10,000 testing images.
- An extended dataset similar to MNIST called EMNIST has been published in 2017, which contains 240,000 training images, and 40,000 testing images of handwritten digits and characters

2.0.1 Import Dataset

[2]:
```
# Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

[3]:
```
# import the MNIST dataset
from keras.datasets import mnist
# load the MNIST dataset
data = mnist.load_data()
```

2.0.2 Training and Testing set

[4]:
```
# Dataset is a class 'tuple'
# A tuple is an (immutable) ordered list of values
print(type(data))
```

```
<class 'tuple'>
```

[5]:
```
# it has two tuples
# data -> data[0] & data[1]
# data[0]-> (train)
# data[1]-> (test)
len(data)
```

[5]: 2

[6]:
```
# each tuple has other two tuples
# each containing: (dataPoints, classLabels)
print(len(data[0]))
print(len(data[1]))
```

```
2
2
```

[7]:
```
# 1st main tuple (train)
# 2nd main tuple (test)
(X_train, y_train), (X_test, y_test) = data
```

Train - X_train: uint8 NumPy array of grayscale image data with shapes (60000, 28, 28), containing the training data. - 60,000 images - Each image is a matrix of 28x28 pixels - Pixel values range from 0 to 255.

- y_train: uint8 NumPy array of digit labels (integers in range 0-9) with shape (60000,) for the training data.

[8]:
```
# shape returns the number of corresponding elements
print(X_train.shape)
print(y_train.shape)
```

```
(60000, 28, 28)
(60000,)
```

Test - X_test: uint8 NumPy array of grayscale image data with shapes (10000, 28, 28), containing the test data. - 10,000 images - Each image is a matrix of 28x28 pixels - Pixel values range from 0 to 255.

- y_test: uint8 NumPy array of digit labels (integers in range 0-9) with shape (10000,) for the test data.

[9]:
```
# shape returns the number of corresponding elements
print(X_test.shape)
print(y_test.shape)
```

```
(10000, 28, 28)
(10000,)
```

2.0.3 Showing the data

[10]:
```
# each data element is a different image
# arranged in a matrix with pixel values range from 0 to 255
# pixels close to 0 tends to black
# pixels close to 255 tends to white
# here the first image
X_train[0]
```

[10]:
```
array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   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,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0]], dtype=uint8)
```

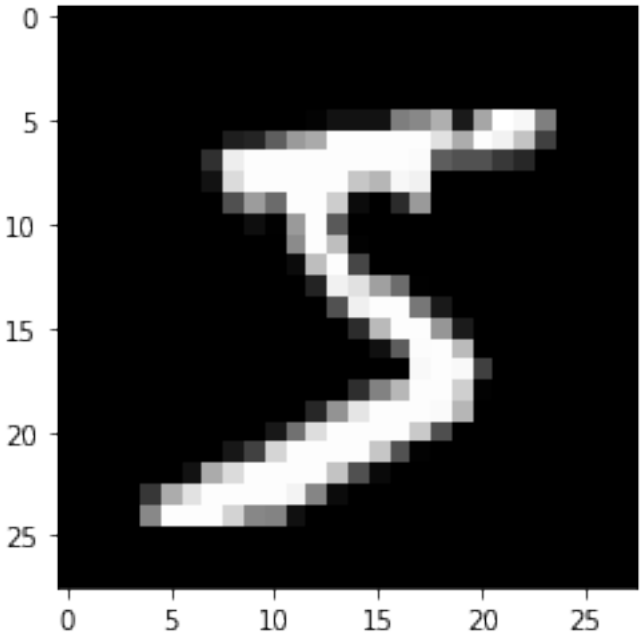
[11]:
```
# class labels are the number
# corresponding to the handwriten

# here the first 10
y_train[0:10]
```

[11]:
```
array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4], dtype=uint8)
```

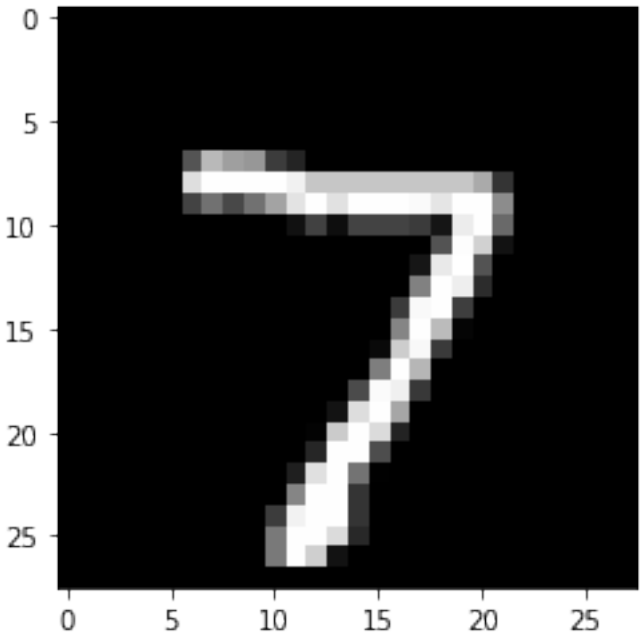
[12]:
```
# A data element in the training set
image_train = 0
plt.imshow(X_train[image_train, :, :], cmap='gray')
print('Class label for this image: ' + str(y_train[image_train]))
```

```
Class label for this image: 5
```



[13]:
```
# A data element in the test set
image_test = 0
plt.imshow(X_test[image_test,:,:],cmap='gray')
print('Class label for this image: ' + str(y_test[image_test]))
```

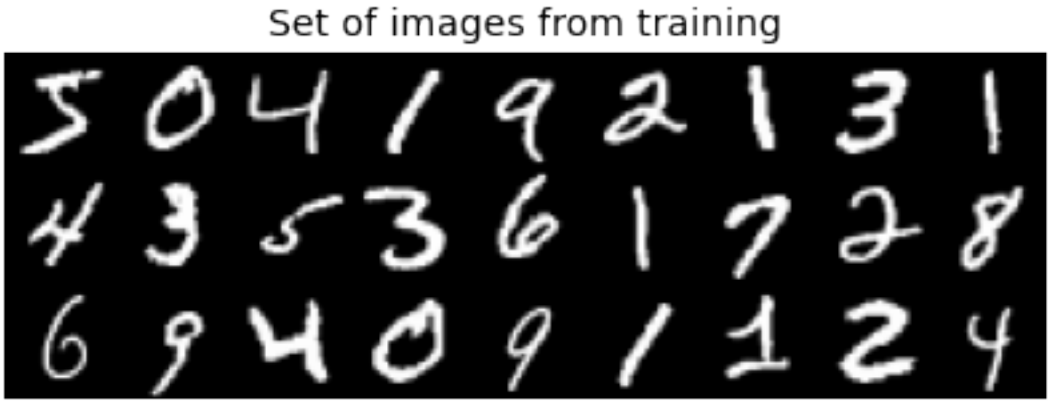
```
Class label for this image: 7
```



[14]:
```
# Function to plot an arrange of images
def plot_images(instances, images_per_row=5, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in instances]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) * images_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image,  cmap='gray', **options)
    plt.axis("off")
```

[15]:
```
# Plotting a set of images from training
plt.figure(figsize=(7, 7))
plot_images(X_train[0: 27, :], images_per_row=9)
plt.title("Set of images from training", fontsize=14)
print("True value =\n", y_train[0:27].reshape(-1, 9))
```

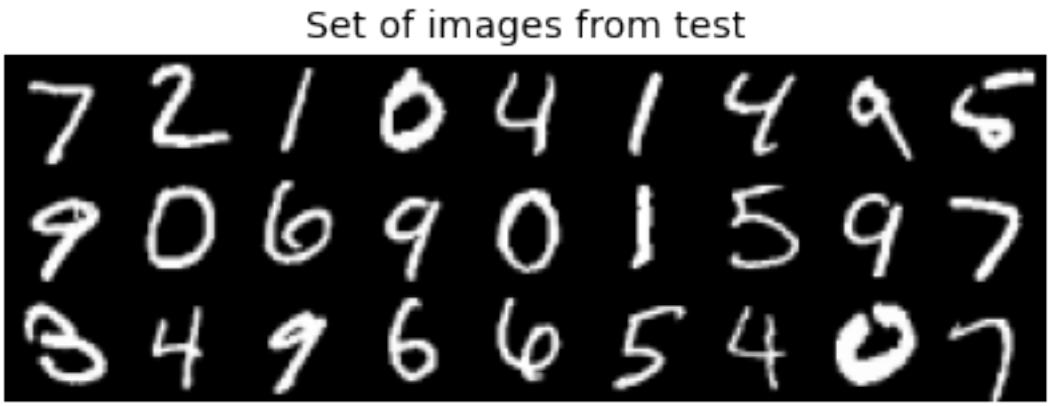
```
True value =
 [[5 0 4 1 9 2 1 3 1]
 [4 3 5 3 6 1 7 2 8]
 [6 9 4 0 9 1 1 2 4]]
```



[16]:
```
# Plotting a set of images from test
plt.figure(figsize=(7,7))
plot_images(X_test[0:27,:],images_per_row=9)
```

```
plt.title("Set of images from test", fontsize=14)
print("True value =\n",y_test[0:27].reshape(-1,9))
```

```
True value =
 [[7 2 1 0 4 1 4 9 5]
 [9 0 6 9 0 1 5 9 7]
 [3 4 9 6 6 5 4 0 7]]
```



3 Classifying images

An image (from Latin: imago) is an artifact that depicts visual perception, such as a photograph or other two-dimensional picture, that resembles a subject—usually a physical object—and thus provides a depiction of it.

- In the context of signal processing, an image is a distributed amplitude of color(s).
- A greyscale image is one in which the value of each pixel is a single sample representing only an amount of light; that is, it carries only intensity information.
- Greyscale images, a kind of black-and-white or grey monochrome, are composed exclusively of shades of grey.
- The contrast ranges from black at the weakest intensity to white at the strongest.

[17]:
```
# Visualize the intensity values
# and the actual tone in each pixel
image2show = 10

df = pd.DataFrame(X_train[image2show, :, :])
df = df.style.background_gradient(cmap='gray')
display(df)

print('\n\n\nClass label for this image: ' + str(y_train[image2show]))
```

```
<pandas.io.formats.style.Styler at 0x7fb0199bb790>
```

```
Class label for this image: 3
```

3.1 Converting a set of images into a table

Convert each image (a 28×28 matrix) into a **row** vector, whose dimension is 1×784.

[18]:
```
X_train_r = X_train.reshape(X_train.shape[0], 28*28)
X_test_r  = X_test.reshape(X_test.shape[0], 28*28)
```

[19]:
```
datav = np.hstack([X_train_r, y_train.reshape(-1, 1)])
df = pd.DataFrame(datav[0:7, :])
df = df.style.background_gradient(cmap='gray')
display(df)
```

```
<pandas.io.formats.style.Styler at 0x7fb0196f6dd0>
```

3.2 Scaling

Scale the resulting matrix to the interval $[0, 1]$, so we can now apply a machine leaning such as: * logistic regression * multi-layer perceptron

[20]:
```
## scale to [0,1]

# dividing over the max value: 255
X_train_r = X_train_r/255
X_test_r = X_test_r/255
```

4 Methods

4.1 Logistic Regression

Logistic Regression, in statistics the logistic model (or logit model) is used to model the probability of a certain class or event existing such as pass/fail, win/lose, alive/dead or healthy/sick.

- This can be extended to model several classes of events such as determining whether an image contains a cat, dog, lion, etc.
- Each object being detected in the image would be assigned a probability between 0 and 1, with a sum of one.

4.1.1 Student Note

The code cell below raises warnings regarding the model not converging before reaching the max iteration limit. I have since updated the max_iter count to be 1000, now the warning is gone.

Upside of having a larger iteration count is the model can have a better chance of reducing the loss, downside is longer training time, and potential risk of overfitting.

[23]:
```
# import the LogisticRegression
from sklearn.linear_model import LogisticRegression
import time

LR = LogisticRegression(multi_class='multinomial',solver='lbfgs', ␣
 ↪fit_intercept=True, max_iter=1000)

print(f'LR model created, training started at {time.time():.2f}')
LR.fit(X_train_r, y_train)

print(f'Training completed at {time.time():.2f}, test starting...')
y_pred = LR.predict(X_test_r)

print(f'Test complete at {time.time():.2f}')
print("Training set score: %f" % LR.score(X_train_r, y_train))
print(f"Testing set score: {LR.score(X_test_r, y_test)}")
```

```
LR model created, training started at 1628107021.32
Training completed at 1628107152.67, test starting…
Test complete at 1628107152.69
Training set score: 0.939267
Testing set score: 0.9256
```

[26]:
```
# Cross check y_pred with y_test actual labels
match_count = 0
for (actual, predict) in zip(y_test, y_pred):
    if actual == predict:
        match_count += 1

print(f'Correct prediction rate: {match_count / len(y_test) * 100:.2f}%')  #␣
 ↪this number is the same as "Testing set score" in the above code cell.
```

```
Correct prediction rate: 92.56%
```

4.2 Multi-layer perceptron

A multilayer perceptron (MLP) is a class of feedforward artificial neural network (ANN).

- The term MLP is used ambiguously, sometimes loosely to any feedforward ANN, sometimes strictly to refer to networks composed of multiple layers of perceptrons.
- An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer.
- Except for the input nodes, each node is a neuron that uses a nonlinear activation function.
- MLP utilizes a supervised learning technique called backpropagation for training.
- Its multiple layers and non-linear activation distinguish MLP from a linear perceptron.

[30]:
```
# import the MLPClassifier
from sklearn.neural_network import MLPClassifier
```

```
MLPC = MLPClassifier(hidden_layer_sizes=(50,), max_iter=500, alpha=1e-4,
                     solver='sgd', verbose=10, tol=1e-4, random_state=1,
                     learning_rate_init=.1)
```

4.2.1 Student Note

The code cell below with training the MLP model was raising Convergence Warning - reaching the max iter count before converging. I have updated the max_iter count to be 500 in the code cell above for when creating the model.

[31]:
```
# train MLPClassifier
MLPC.fit(X_train_r, y_train)

# TODO: try fix the sklearn convergence warning
```

```
Iteration 1, loss = 0.32009978
Iteration 2, loss = 0.15347534
Iteration 3, loss = 0.11544755
Iteration 4, loss = 0.09279764
Iteration 5, loss = 0.07889367
Iteration 6, loss = 0.07170497
Iteration 7, loss = 0.06282111
Iteration 8, loss = 0.05530788
Iteration 9, loss = 0.04960484
Iteration 10, loss = 0.04645355
Iteration 11, loss = 0.04082169
Iteration 12, loss = 0.03828222
Iteration 13, loss = 0.03557957
Iteration 14, loss = 0.03054891
Iteration 15, loss = 0.02924761
Iteration 16, loss = 0.02610471
Iteration 17, loss = 0.02363894
Iteration 18, loss = 0.02208186
Iteration 19, loss = 0.01932900
Iteration 20, loss = 0.01830387
Iteration 21, loss = 0.01639227
Iteration 22, loss = 0.01392950
Iteration 23, loss = 0.01270193
Iteration 24, loss = 0.01234102
Iteration 25, loss = 0.01081313
Iteration 26, loss = 0.01028644
Iteration 27, loss = 0.00896707
Iteration 28, loss = 0.00744908
Iteration 29, loss = 0.00707946
Iteration 30, loss = 0.00573869
Iteration 31, loss = 0.00499554
Iteration 32, loss = 0.00477064
Iteration 33, loss = 0.00395458
```

```
Iteration 34, loss = 0.00355619
Iteration 35, loss = 0.00375497
Iteration 36, loss = 0.00304228
Iteration 37, loss = 0.00264245
Iteration 38, loss = 0.00241425
Iteration 39, loss = 0.00234957
Iteration 40, loss = 0.00233803
Iteration 41, loss = 0.00204653
Iteration 42, loss = 0.00199057
Iteration 43, loss = 0.00190567
Iteration 44, loss = 0.00180530
Iteration 45, loss = 0.00175054
Iteration 46, loss = 0.00168160
Iteration 47, loss = 0.00162517
Iteration 48, loss = 0.00159676
Iteration 49, loss = 0.00154993
Iteration 50, loss = 0.00152799
Iteration 51, loss = 0.00146697
Iteration 52, loss = 0.00145257
Iteration 53, loss = 0.00143422
Iteration 54, loss = 0.00135888
Iteration 55, loss = 0.00134281
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs.
Stopping.
```

[31]:
```
MLPClassifier(hidden_layer_sizes=(50,), learning_rate_init=0.1, max_iter=500,
              random_state=1, solver='sgd', verbose=10)
```

4.2.2 Student Note

The training stopped at iteration 55 as there wasn't anymore significant reduction in loss.

[32]:
```
print("Training set score: %f" % MLPC.score(X_train_r, y_train))
```

```
Training set score: 1.000000
```

4.2.3 Student Note

Training set score of 100% is a bit concerning, let's see how the model behaves with the testing set.

[33]:
```
print(f'Testing set score: {MLPC.score(X_test_r, y_test)}')
```

```
Testing set score: 0.9731
```

4.2.4 Student Note

97.31% is pretty good! Happy days.

5 Tasks:

- Compute the accuracy of the classifier
- Compute the confusion matrix of the predictions versus the true classes (see link about what a confusion matrix is and here how to compute and plot it).
- Visualise (using the plot function provided at the beginning of the notebook) instances (images) where the predicted class was wrong. What can you notice?

5.1 Task 1 - Compute the accuracy of the classfiers

[57]:
```
# Accuracy of the Logistic Regression model
# This LR.score method will take in the test dataset, test labels,
#  run the model's prediction, then compare the prediction result with the␣
 ↪actual labels,
#  finally return a match percentage.
LR.score(X_test_r, y_test)
```

[57]: 0.9256

[35]:
```
# Accuracy of the MLP model
MLPC.score(X_test_r, y_test)
```

[35]: 0.9731

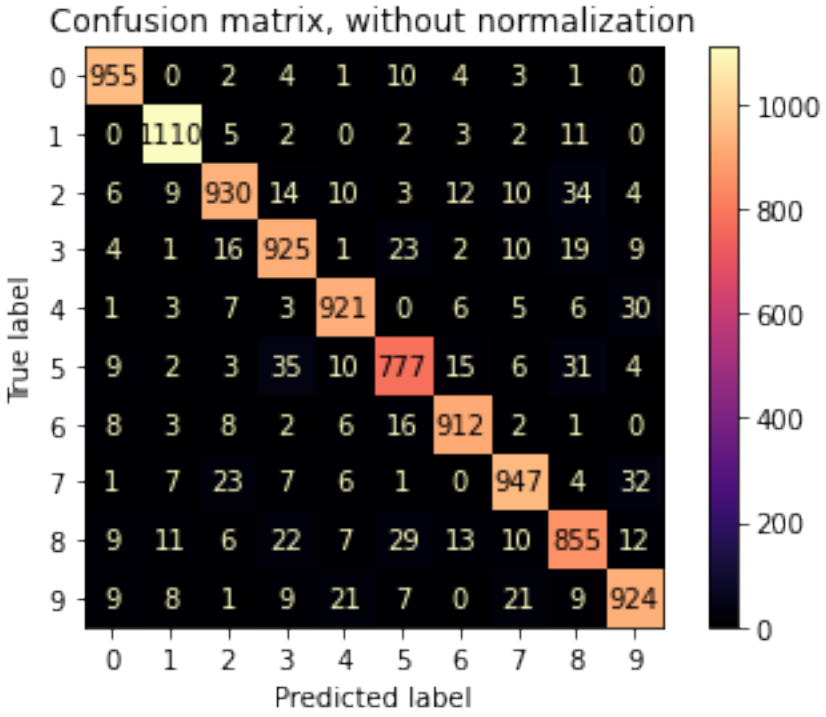
5.2 Task 2 - Compute the confusion matrix of the predictions versus the true classes

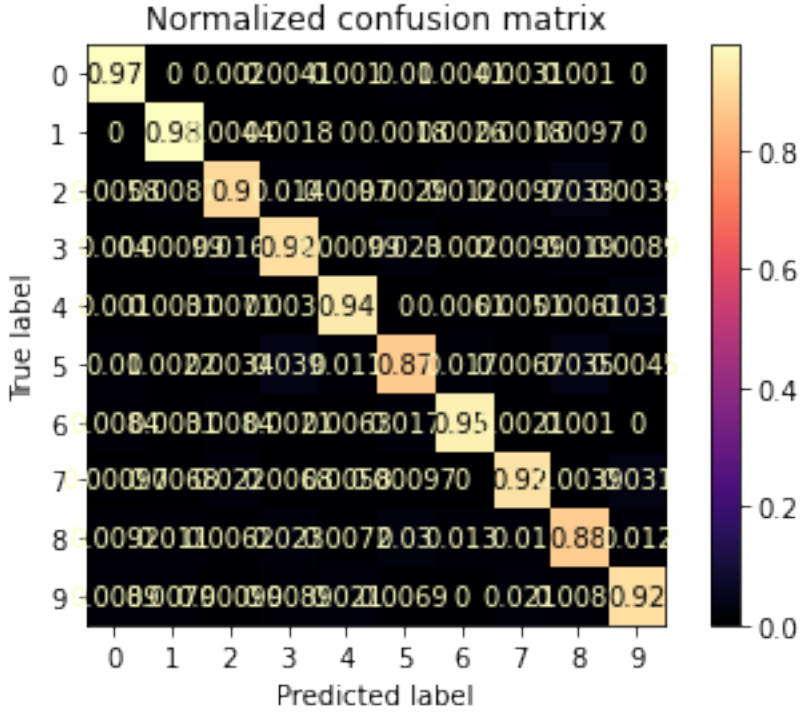
5.2.1 Logistic Regression

[58]:
```
# Metrics plot preperation
from sklearn.metrics import plot_confusion_matrix
# The classes we have in the dataset is hand written digits between 0 - 9, put␣
 ↪these into a list
classes = [i for i in range(0, 10)]
# this titles_options is later used in the plot_confusion_matrix method,
# for plotting a raw count, and a normalized value.
titles_options = [("Confusion matrix, without normalization", None),
                  ("Normalized confusion matrix", 'true')]
```

[62]:
```
# Iterate through non-normalized, and normalized plotting options
for title, normalize_flag in titles_options:
    # call the plot_confusion_matrix method with the model (LR), test data set,␣
 ↪and its labels
    display = plot_confusion_matrix(LR, X_test_r, y_test,
                                    display_labels=classes,  # classes from 0 to 9
                                    cmap='magma',  # colour map used in the␣
 ↪confusion matrix chart
                                    normalize=normalize_flag)
```

```
    display.ax_.set_title(title)
```





5.2.2 Student Note

From the confusion matrix above, we can see that for class '1', the model is predicting very well (as can be seen from the bright yellow colour and its normalized value of 0.98).

For class '5', the model is struggling the most, with the lowest match count of 777 and a normalized value of 0.87.

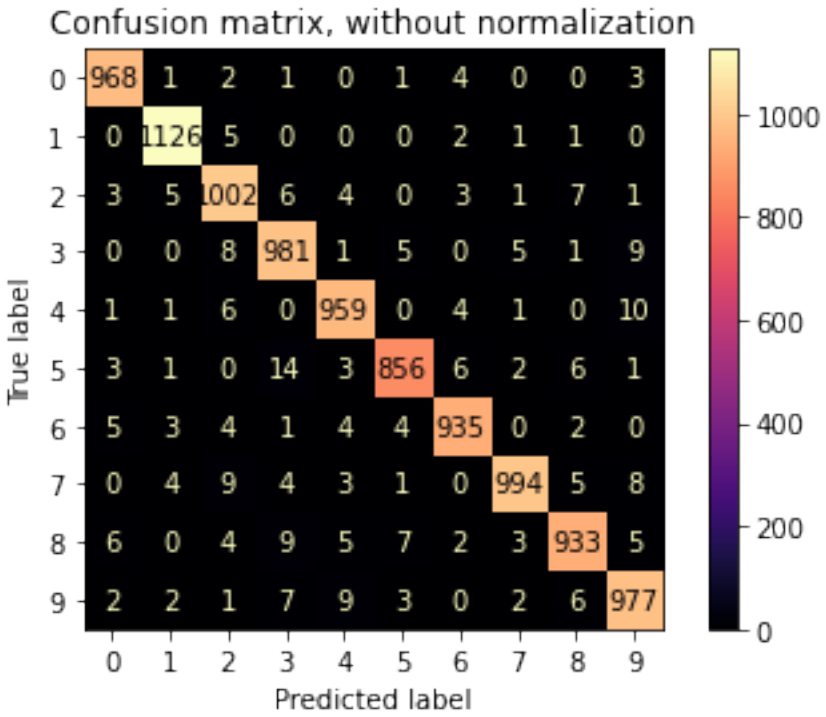
If we look at the horizontal line where True label is 5, we can see the count for prediction '3' and '8' is higher than others, this is most likely due to the fact that people write number 5 and sometimes it looks like 3 or 8.

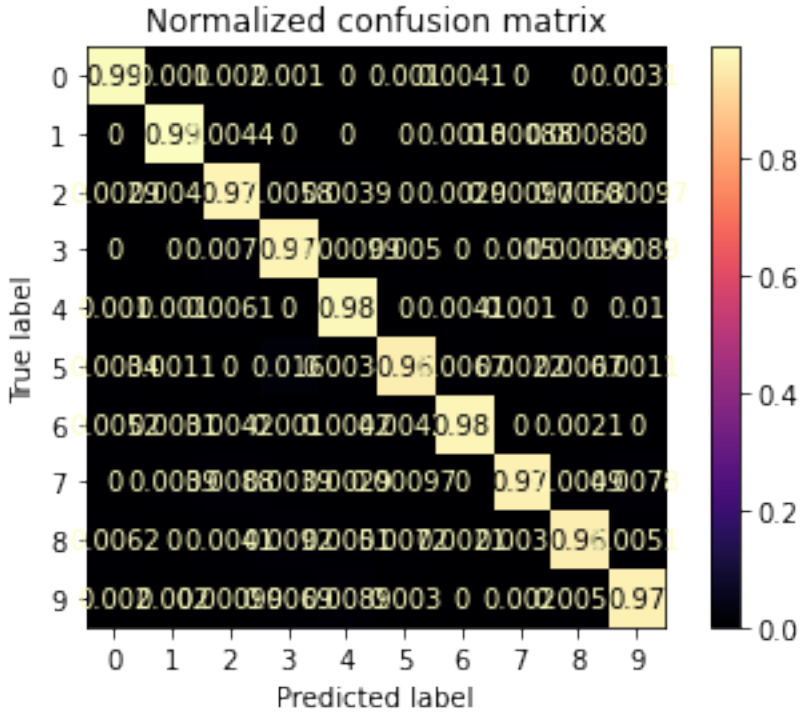
5.2.3 Student Note

See this link for a full list of colour map you can choose for plotting the confusion matrix.

5.2.4 Multi-Layer Perceptron

[63]:
```
for title, normalize_flag in titles_options:
    display = plot_confusion_matrix(MLPC, X_test_r, y_test,  # this time pass␣
↪in the MLP model, the rest is the same.
                                    display_labels=classes,
                                    cmap='magma',
                                    normalize=normalize_flag)
    display.ax_.set_title(title)
```





5.2.5 Student Note

For the MLP model, we can see the overall performance is considerably better than the LR model, with '1' being the easiest class to predict, while all classes have a normalized score of 0.96 and above.

5.3 Task 3 - Visualise instances (images) where the predicted class was wrong. What can you notice?

5.3.1 Logistic Regression

[77]:
```
# First, find the instances that were mis-classified by the LR model

# Run the prediction and store results into LR_prediction
LR_prediction = LR.predict(X_test_r)

# An empty list to hold the indices of the mis-classified instances
LR_mistake_indices = []

# Go through the actual labels and the prediction result, compare the two,
# if there is a mismatch, append this instance's index to the list above
for i, (true_label, prediction) in enumerate(zip(y_test, LR_prediction)):
```

```
    if true_label != prediction:
        LR_mistake_indices.append(i)

print(len(LR_mistake_indices))
```

```
744
```

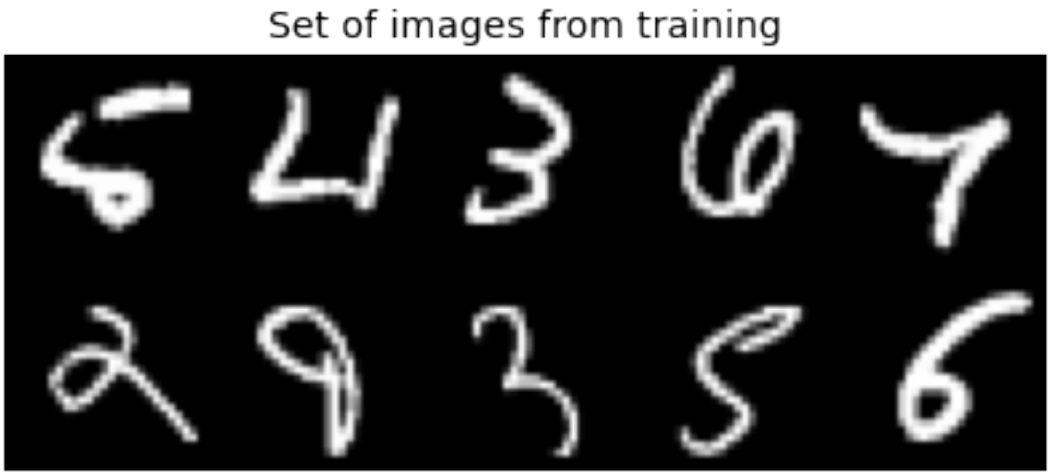
5.3.2 Student Note

There are 744 instances where the LR model got it wrong. Let's take a look at the first 10

[84]:
```
# Plotting a set of images from the original test set before reshaping
plt.figure(figsize=(7, 7))

# here we can access certain items in a list (X_test) by passing another list␣
↪(first 10 of LR_mistake_indices),
# this loosely translates to "from this X_test list, give me the items at these␣
↪indices"
# This same code is used in a code cell below for accessing the labels.
plot_images(X_test[LR_mistake_indices[:10]])
plt.title("Set of images from training", fontsize=14)
```

[84]:
```
Text(0.5, 1.0, 'Set of images from training')
```



5.3.3 Student Note

These images don't look like numbers!!! Written by a spider on a hot plate :)

5.3.4 Let's see what they actually are, and what the model predicted them as

[75]:
```
print("True value =\n", y_test[LR_mistake_indices[:10]].reshape(2, 5))
print("Prediction =\n", LR_prediction[LR_mistake_indices[:10]].reshape(2, 5))
```

```
True value =
 [[5 4 3 6 7]
 [2 9 3 5 6]]
Prediction =
 [[6 6 2 2 4]
 [9 3 5 7 5]]
```

5.3.5 Multi-Layer Perceptron

[78]:
```
# First, find the instances that were mis-classified by the LR model

# Run the prediction and store results into LR_prediction
MLP_prediction = MLPC.predict(X_test_r)

# An empty list to hold the indices of the mis-classified instances
MLP_mistake_indices = []

# Go through the actual labels and the prediction result, compare the two,
# if there is a mismatch, append this instance's index to the list above
for i, (true_label, prediction) in enumerate(zip(y_test, MLP_prediction)):
    if true_label != prediction:
        MLP_mistake_indices.append(i)

print(len(MLP_mistake_indices))
```

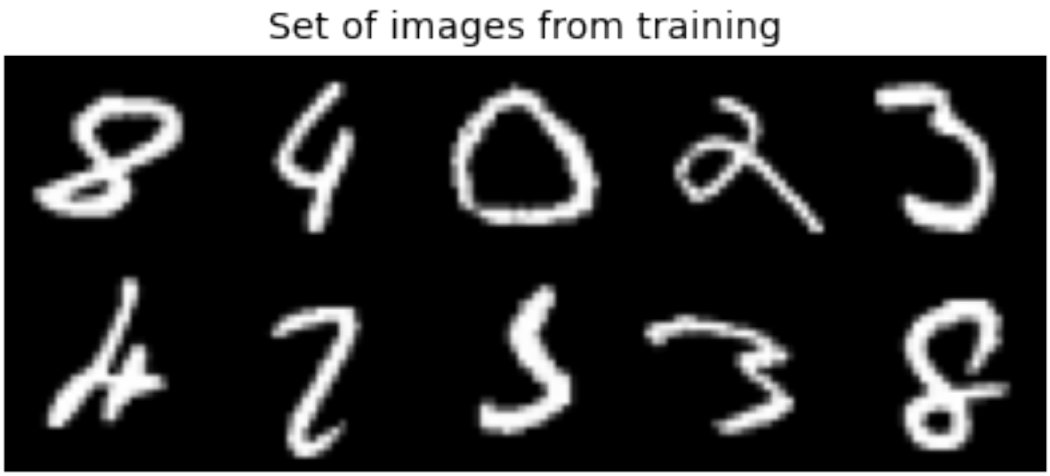
```
269
```

5.3.6 Student Note

MLP model got 269 instances wrong, much better than the LR model. Let's take a look

[79]:
```
# Plotting a set of images from the original test set before reshaping
plt.figure(figsize=(7, 7))
plot_images(X_test[MLP_mistake_indices[:10]])
plt.title("Set of images from training", fontsize=14)
```

[79]:
```
Text(0.5, 1.0, 'Set of images from training')
```



5.3.7 Student Note

Yep, let's take a look at the actual labels and the prediction result for these ancient Rome wall art

[82]:
```
print("True value =\n", y_test[MLP_mistake_indices[:10]].reshape(2, 5))
print("Prediction =\n", MLP_prediction[MLP_mistake_indices[:10]].reshape(2, 5))
```

```
True value =
 [[8 4 0 2 3]
 [4 2 5 3 8]]
Prediction =
 [[2 9 9 4 2]
 [2 7 3 7 2]]
```

6 Summary

Some of these hand written digits don't look like any numbers between 0 - 9, or, one could say they look like anything. Perhaps a higher resolution of the same images would improve the model's accuracy?

In terms of overall performance, the Multi-Layer Perceptron model is considerably better than the Logistic Regression model, with 97% accuracy on the test set, while LR has an accuracy of about 92% on the same test set.

The orignal settings of 10 for max iteration for both models were too small, the models couldn't converge, raising them helped the model's convergence and raised the accuracy (reduced loss).

Sci-kit Learn's prebuilt confusion matrix plotting is a great little tool!

Also thanks to Enrique's method for organizing and plotting the images, it's a very neat way of putting a list of images into a grid.

7 References

1. Confusion Matrix from Sci-kit Learn.