

Python Practical Session on the Recognition of MNIST Handwritten Digits

Instructors

Battista Biggio and **Luca Didaci**

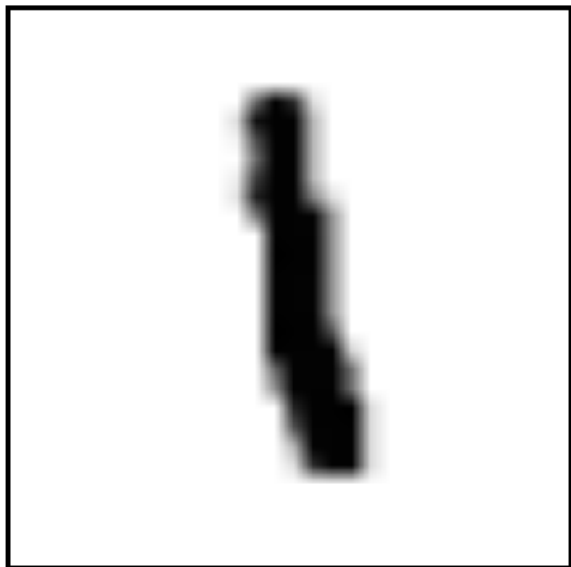
M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

MNIST Handwritten Digit Dataset

- Popular benchmark dataset for image classification
 - Y. LeCun, L. Jackel, L. Bottou, A. Brunot, C. Cortes, J. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard, and V. Vapnik. Comparison of learning algorithms for handwritten digit recognition. In Int'l Conf. on Artificial Neural Networks, pp. 53–60, 1995.
- Handwritten digit images (centered and normalized as 28x28 images) belonging to 10 different classes (digits from 0 to 9)
- Each image \mathbf{x} is stored as a flat vector of $28 \times 28 = 784$ values (gray-level value of each pixel)
- The class label of each digit is also provided



MNIST Handwritten Digit Dataset

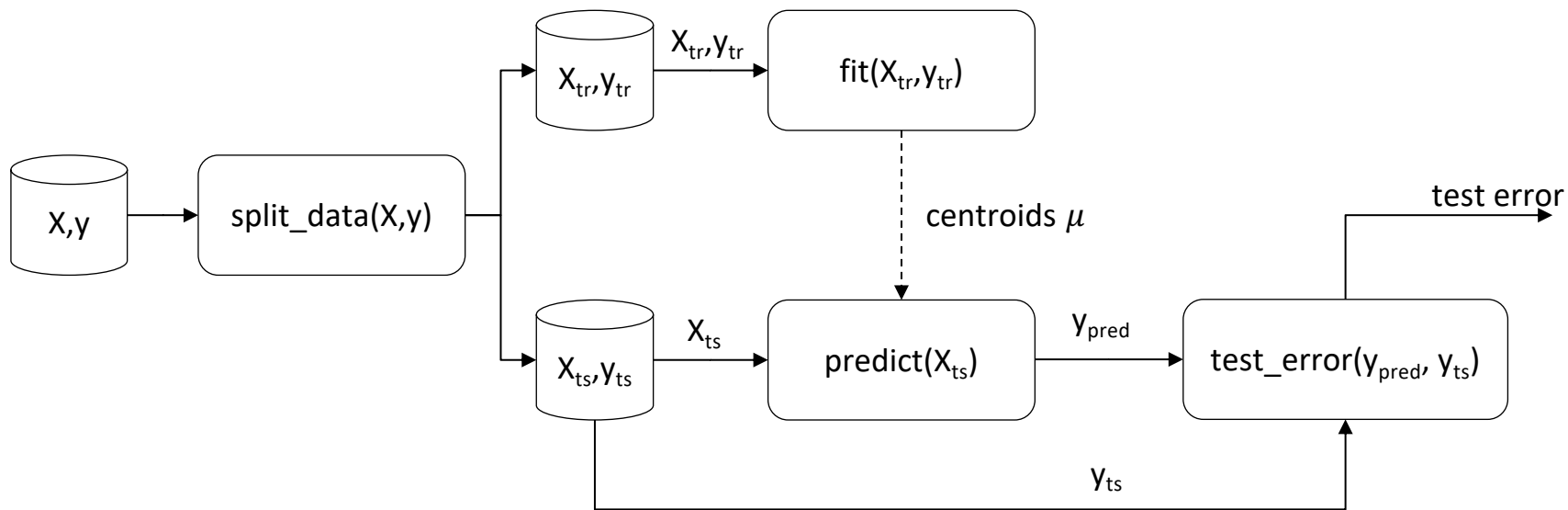


12

0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	.6	.8	0	0	0	0	0	0
0	0	0	0	0	0	.7	1	0	0	0	0	0	0
0	0	0	0	0	0	.7	1	0	0	0	0	0	0
0	0	0	0	0	0	.5	1	.4	0	0	0	0	0
0	0	0	0	0	0	0	1	.4	0	0	0	0	0
0	0	0	0	0	0	0	1	.4	0	0	0	0	0
0	0	0	0	0	0	0	1	.7	0	0	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0	0
0	0	0	0	0	0	0	.9	1	.1	0	0	0	0
0	0	0	0	0	0	0	.3	1	.1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Coding Exercise

- We have to build a simple system that recognizes the class of digit images
- Its architecture is depicted below

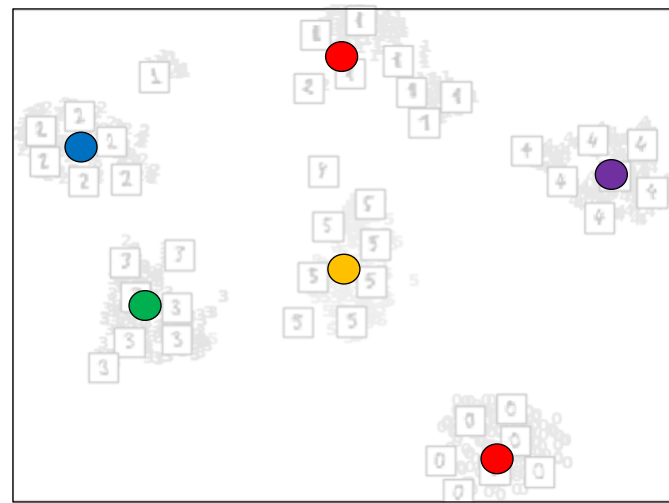
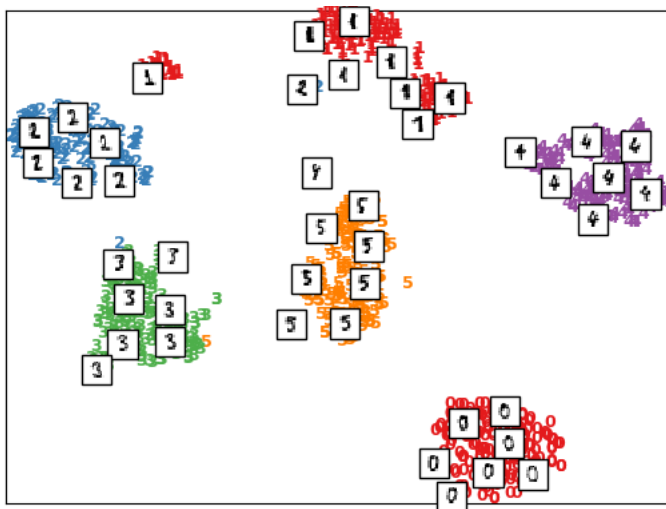


Coding Exercise

- The system should work as follows. Given the digit images as rows of X (namely, a matrix with $num_samples$ rows and $num_dimensions = 784$ columns), and their labels as a vector y of $num_samples$ elements:
 1. Split the input data X, y into a training (X_{tr}, y_{tr}) and a test set (X_{ts}, y_{ts}) by randomly selecting a fraction of images from X and labels from y to be part of the training set, while assigning the remaining ones to the test set;
 2. Compute the average image (centroid) for each class from the training data (fit), and display the centroids (average digits) in a plot;
 3. Compute the distance of each test image in X_{ts} against all centroids, and predict the label of the current sample y_{pred} as that of the closest centroid (predict);
 4. Evaluate the test error, namely, the fraction of misclassified test images (for which $y_{pred} \neq y_{ts}$).
- This is your very first, simple implementation of the machine-learning algorithm known as Nearest Mean Centroid (NMC) classifier.

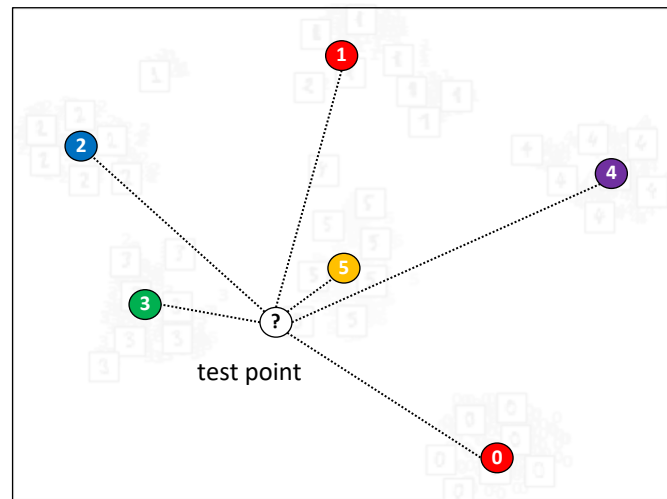
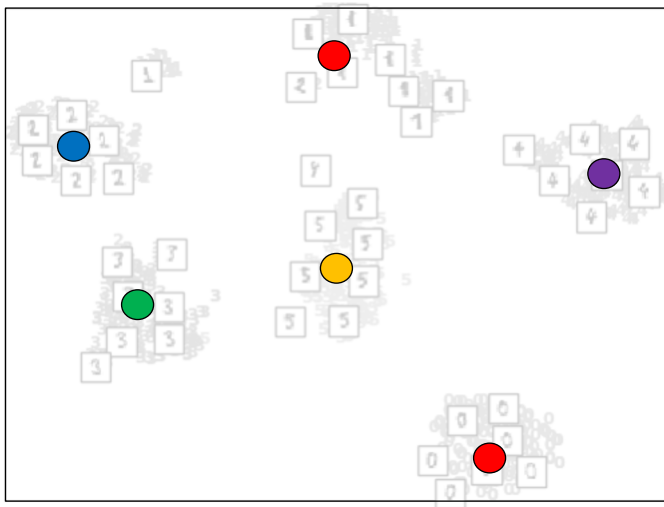
NMC Classifier: «fit»

- Each digit image can be represented as a point in a vector space where each dimension corresponds to the value of each pixel
 - we can plot training digits of different classes with different colors / markers
 - *fit* estimates the average image (centroid) for each class



NMC Classifier: «*predict*»

- *Predict* computes the Euclidean distance of a given test point against all centroids, and assigns it to the class of the closest centroid
 - The test point ('?') below is classified as a '5', as it is closer to the centroid of class '5'
 - The length of each dashed line is the distance between the test point and the given centroid



Why Training and Test Data?

- The idea of using separate training and test data is motivated by the need of testing the performance (*test error*) of the learning algorithm on never-before-seen data, namely, data which has not been used to learn its parameters (i.e., the centroids, in our case).
- Measuring the performance of the classifier on the same data used for training is wrong and too optimistic: the learning algorithm has already been shown such images while estimating the centroids!
- *We will see more in the Machine Learning course*

Solution

The solution code (implementation) is provided separately
Here only the key algorithmic details are explained

Load Data

- `load_data(filename)` simply uses `read_csv` from *pandas* to load data from a file
- The file contains the class labels in the first column, and the digit images as rows
 - In practice, each row contains the class label and then 784 gray-level pixel values (the image)
- We thus extract the class labels y from the first column, and the digit images X as the remaining data
 - y is simply a numpy array of $(num_samples,)$
 - X is a numpy array of $(num_samples, num_pixels)$

```
def load_data(filename):  
    """  
    Load data from a csv file  
  
    Parameters  
    -----  
    filename : string  
        Filename to be loaded.  
  
    Returns  
    -----  
    X : ndarray  
        the data matrix.  
  
    y : ndarray  
        the labels of each sample.  
    """  
    data = read_csv(filename)  
    z = np.array(data)  
    y = z[:, 0]  
    X = z[:, 1:]  
    return X, y
```

Split Data

- This function has to split X, y into a training and a test set
- The training set has to include $tr_fraction$ elements (in %)
- We first compute n_tr : how many samples should go in the training set
- Then, we create a vector of indices idx , and shuffle it. The first n_tr indices will be the indices of the training samples, while the remaining indices will be used to select the test samples.

```
def split_data(X, y, tr_fraction=0.6):  
    """  
    Split the data X,y into two random subsets  
  
    """  
    num_samples = y.size  
    n_tr = int(num_samples * tr_fraction)  
  
    idx = np.array(range(0, num_samples))  
    np.random.shuffle(idx) # shuffle indices  
  
    tr_idx = idx[0:n_tr]  
    ts_idx = idx[n_tr:]  
  
    Xtr = X[tr_idx, :]  
    ytr = y[tr_idx]  
  
    Xts = X[ts_idx, :]  
    yts = y[ts_idx]  
  
    return Xtr, ytr, Xts, yts
```

Fit: Compute the Centroids

- Computes the average centroid image for each class from the training data
 - *centroids* is a matrix of 10 rows x 784 columns, as it needs to store 1 centroid (784 values) per class (and we have 10 classes)
- First, we initialize *centroids* (a matrix of zeros)
- Then, we iterate over the classes, and extract the samples x_k belonging to class k
 - `ytr==classes[k]` returns a boolean vector (true if the label in *ytr* is equal to the label of class k) which can be used to select the samples of class k from *Xtr*
- These samples are then averaged using `np.mean` along the correct axis to obtain the corresponding centroid, which is stored as row k in *centroids*

```
def fit(Xtr, ytr):  
    """  
    Compute the average centroid  
    for each class  
    """  
    classes = np.unique(ytr)  
    num_classes = classes.size  
  
    centroids = np.zeros(  
        shape=(num_classes, Xtr.shape[1]))  
  
    for k in range(num_classes):  
        xk = Xtr[ytr == classes[k], :]  
        centroids[k, :] = np.mean(xk, axis=0)  
  
    return centroids
```

Predict Classes of Unseen Test Data

- *Predict* computes a matrix *dist_euclidean* of size $n_{ts} \times 10$, being n_{ts} the number of test samples in *Xts*, which contains the values of the Euclidean distance computed between each test sample and the 10 class centroids
- Then, the index of the closest centroid for each test sample is retrieved with `np.argmin`
 - if the class labels are contiguous (0,1,2,...9), then these indices are already the class labels *ypred* and can be returned (if *classes* is `None`, we assume that this is the case)
 - otherwise, we use them to index the class labels (passed as the input parameter *classes*) and return them

```
def predict(Xts, centroids, classes=None):  
    """  
    Predicts the label of each sample in Xts based on  
    the closest centroid.  
    """  
    dist_euclidean = euclidean_distances(Xts, centroids)  
    ypred = np.argmin(dist_euclidean, axis=1)  
    if classes is not None:  
        ypred = classes[ypred]  
    return ypred
```

Have a look at the code too. There is a more complex implementation of this function where the distance matrix is computed explicitly – not using a library function!

Compute the Test Error

- This function just counts the fraction of different elements between the predicted labels `ypred` and the true labels `yts`
 - Cast to float is required to avoid integer division (which would return 0 or 1 in this case)

```
def compute_ts_error(ypred, yts):  
    """  
    Compute the fraction of elements that  
    are different in ypred and yts  
    (classification errors)  
  
    Parameters  
    -----  
    ypred: the set of predicted class labels  
    yts: the true labels of test samples  
  
    Returns  
    -----  
    test_error: the classification error  
    """  
    test_error = np.sum(ypred != yts) / float(ypred.size)  
    return test_error
```

From functions to classes: the NMC classifier class

Implementation of the NMC classifier as a Python class

The NMC Classifier as a Python Class

- Finally, the previous functions can be put together in a class, representing our learning algorithm.
- `_centroids` is a protected attribute that will store the centroids (the corresponding property allows reading their values from the outside)
- `fit` and `predict` slightly change to use the internal variable `_centroids` (rather than returning the centroids and taking them as input)

```
class NMC(object):  
    """Nearest Mean Centroid (NMC) classifier."""  
  
    def __init__(self):  
        self._centroids = None  
        self._classes = None # class labels may not be contiguous  
  
    @property  
    def centroids(self):  
        return self._centroids  
  
    def fit(self, Xtr, ytr):  
        self._classes = np.unique(ytr)  
        num_classes = self._classes.size  
        self._centroids = np.zeros(shape=(num_classes, Xtr.shape[1]))  
        for k in xrange(num_classes):  
            xk = Xtr[ytr == self._classes[k], :]  
            self._centroids[k, :] = np.mean(xk, axis=0)  
  
    def predict(self, Xts):  
  
        if self._centroids is None:  
            raise ValueError("The classifier is not trained. Call fit!")  
  
        dist_euclidean = euclidean_distances(Xts, self._centroids)  
        idx_min = np.argmin(dist_euclidean, axis=1)  
        yc = self._classes[idx_min]  
        return yc
```