Naïve Bayes Classifier - Lab Exercise

The objective of today's exercise session is to implement a classifier of handwritten digits. We can think of that as a *black box* that receives an image as input and produces an integer ranging from 0 to 9 as output (*i.e.* the predicted class). The dataset we refer to is MNIST, a database consisting of 70000 images (of which 60000 are usually employed to train a model and 10000 to evaluate its perfromance on a never-before-seen set of images, called a test set).

To utilize the dataset we must first load it in memory (this is accomplished by the `load_mnist()` function, which is already provided to you)

```
import numpy as np
import matplotlib.pyplot as plt

x_train, y_train, x_test, y_test, label_dict = \
             load_mnist(which_type='digits', threshold=0.5)
```

As input, the function also gets a `threshold` value; the latter is used internally to binarize all pixels of the greyscale images of MNIST, which are thus

turned into either 0 or 1, depending on their original value being higher or lower than the provided threshold.

```
print(f"Training set -> number of examples: {len(x_train)}")
print(f"Test set -> number of examples: {len(x_test)}")
print('-'*30)
print(f'X -> shape: {x_train.shape}')
print(f"X -> dtype: {x_train.dtype}")
print(f"X -> min: {x_train.min()}")
print(f"X -> max: {x_train.max()}")
print(f"X -> values: {np.unique(x_train)}")
print('-'*30)
print(f"Classes: {(np.unique(y_train))}")
```

In this code fragment, the program prints useful information about the dataset (how many items make up its training and test set, their shape, the data type of pixels, etc.). This produces the following result:

```
Training set -> number of examples: 60000
Test set -> number of examples: 10000
------------------------------
X -> shape: (60000, 28, 28)
X -> dtype: float32
X -> min: 0.0
X -> max: 1.0
X -> values: [0. 1.]
------------------------------
Classes: [0 1 2 3 4 5 6 7 8 9]
```

The training set is simply a tensor (a Numpy array) of type `float32`, with shape $(60000, 28, 28)$: *i.e.,* it's made up of 60000 examples, each of which with shape $28 \cdot 28$. While implementing the proposed solution, the dimensions will be re-organized (*reshaping*) for a matter of convenience: hence, we'll make our $(60000, 28, 28)$-shaped object into a $(60000, 784)$ tensor (in the latter, each example is a 1d array composed of 784 features). This operation does not influence the content of any example in the training set, we're just changing its representation from a matrix (first case) to a vector (latter case).

Before we analyze the content of the `NaiveBayesClassifier()` class (whose main concern is implementing the training and prediction logic of the classifier), let us examine the content of the sixth and seventh cell in the solution:

```
# get the model
nbc = NaiveBayesClassifier()

# train
nbc.fit(x_train, y_train)

# test
```

```
predictions = nbc.predict(x_test.reshape((len(x_test), -1)))

# evaluate performances
accuracy = np.sum(np.uint8(predictions == y_test)) / len(y_test)
print('Accuracy: {}'.format(accuracy))

# show confusion matrix
plot_confusion_matrix(targets=y_test,
                      predictions=predictions,
                      classes=[label_dict[l] for l in label_dict])
```

We will encounter this code fragment very often in our future lab sessions:

1. `nbc = NaiveBayesClassifier()` →a classifier object is initialized. Typically, the contructor receives as input some hyper-parameters (none here).
2. `nbc.fit(x_train, y_train)` →the classifier is trained on the training set (of course, we need both the images and their corresponding labels for this step).
3. `predictions = nbc.predict(x_test.reshape((len(x_test), -1)))` →**inference**: we obtain the classifier's predictions on the test data.
4. `accuracy = np.sum(np.uint8(predictions == y_test)) / len(y_test)` →we compare the model's predictions with the labels of the test set. This is done through specific metrics (here, we use accuracy – *i.e.* the number of correctly classified examples over the total – and the confusion matrix).

Let us now move on to the core of the exercise: implementing `NaiveBayesClassifier()`. This is the classifier's prediction rule:

$$\hat{y} = \underset{k \in 1,\dots K}{\operatorname{argmax}} p(y_k) \prod_{i=1}^{n} p(x_i|y_k), \tag{1}$$

that's to say, the predicted class $\hat{y}$ is the one that maximizes the product of the following:

1. **prior** →$p(y_k)$ the probability of observing an example of class y_k. This is called prior because it does not depend on the observed example x.
2. **likelihood** →$p(x|y_k)$ (which, thanks to the naïve hypothesis can be rewritten as the products of the individual components (*i.e.* pixels) – $\prod_{i=1}^{n} p(x_i|y_k)$. This quantity tells us how likely we are to observe a given example x within a given class y_k.

In the training step, we must examine the training set and use it to estimate the prior of each class and along with its "likelihood model" (which, in the example at hand, is a vector for each class, containing as many items a as pixels in an image, that keeps track of the probability of a given pixel being on or off).

These are the two fundamental data structures that we must produce, they are defined for you in the last two instructions of the class constructor.

```
lass NaiveBayesClassifier:

    def __init__(self):
        """
        Class constructor
        """

        self._classes = None
        self._n_classes = 0

        self._eps = np.finfo(np.float32).eps

        # array of classes prior probabilities
        self._class_priors = []

        # array of probabilities of a pixel being active (for each class)
        self._pixel_probs_given_class = []
```

The `fit(self, X, Y)` will simply examine the training set and:

1. Establish what is the set of classes (in our case, integers between 0 and 9).
2. Establish the prior for each class: this can be accomplished by calculating how often each class appears in the training set (which can be done by using the function `np.unique(Y, return_counts=True)`) and then by dividing by the total number of examples.
3. Compute for each class k its likelihood model. That is, the vector that contains for each of the 784 pixels its probability of being on within class k.

```
def fit(self, X, Y):

        yclass, counts = np.unique(Y, return_counts=True)

        self._classes = yclass
        self._n_classes = len(yclass)
        self._class_priors = counts / X.shape[0]

        for i in range(self._n_classes):
            pixel_prob_given_i = np.mean(X[Y == i], axis=0)
            self._pixel_probs_given_class.append(pixel_prob_given_i)
```

The computation of the likelihood model happens at **line 10** in the code above.

1. `Y == i` produces a vector of bools (with the same shape as Y (60000,) – a boolean for each example): `True` if the corresponding example belongs to the i^{th} class, `False` otherwise.

2. `X[Y == i]` selects from X all examples such that `Y == i` is true – *i.e.*, it selects all examples of class i in X.

3. `np.mean(X[Y == i], axis=0)` reduces all examples of class i using the mean. By so doing, I quickly obtain the probability of each pixel being on within the class. Alternatively, we could compute the how many times this pixels is on and divide it by the total... but, if you stop to think about it, that is exactly the same as computing the average of all examples!

We can now move on to the inference phase. For the sake of numerical stability, we do not use simple probabilities as written in Eq. 1, but rather log-probs:

$$\hat{y} = \underset{k \in 1,\dots K}{\operatorname{argmax}} \log p(y_k) + \sum_{i=1}^{n} \log p(x_i | y_k). \tag{2}$$

The `predict(self, X)` method is concerned with associating to each input example its predicted class. It is structured as follows:

1. `results = np.zeros((n_test_images, self._n_classes))` prepares a matrix that we will use to store the score of each class for each example. The following loop properly fills up the values of this matrix.

2. for each class, we compute how much each example adheres to its likelihood model. As a final step, we add up the log-prob associated to the prior.

```
def predict(self, X, return_pred: bool = False):

        n_test_images = X.shape[0]

        X = X.reshape((n_test_images, -1))
        results = np.zeros((n_test_images, self._n_classes))

        for i in range(self._n_classes):
            # compute log P(X|y=i)
            model_of_i = self._pixel_probs_given_class[i]
            model_of_i = model_of_i.reshape((1, X.shape[1]))

            mask_one = X == 1.0
            mask_zero = X == 0.0

            probs = mask_one * model_of_i + mask_zero * (1. - model_of_i)
            probs = np.log(probs + self._eps)
            probs = np.sum(probs, axis=1)

```

```
20              probs += np.log(self._class_priors[i])
21              results[:, i] = probs
22
23          if not return_pred:
24              return np.argmax(results, axis=1)
25          return np.argmax(results, axis=1), results
```

The adherence of examples to likelihood models is computed at lines 10-18. The gist of it is: given an exmample x and one of its pixels x_j one of two things can happen:

1. $x_j = 1$ $\rightarrow$in this case, the value of $p(x_j|y_k)$ is equal to what we have in the likelihood model at position j.

2. $x_j = 0$ $\rightarrow$in this case, we must recover from the likelihood model the probability of observing pixel j **in an off state**; the latter is obtained as $1 - p_j$.