1 Implementation of Perceptron

Perceptron is a linear classifier used for binary classification, that assumes that the data is linearly separable. If this holds, then the model is always able to find a decision boundary which is an hyperplane of kind:

$$H = \{x : \hat{w}^T x + b = 0\}$$

able to linearly separate the two classes of our binary problem.
We can then absorb the bias term b into w (and accordingly add one feature containing "1") to x and obtain an expression for classifying our data of kind:

$$y_i = h(x_i) = sign(w^T x_i)$$

1.1 The train function

What we need to find is the hyperplane defined by w.In order to do that we need to minimize the **perceptron criterion** $E_p(w) = -\sum_{i \in M} w^T x_i y_i$ where M is the set of misclassified points[1]. To minimize the perceptron criterion, we used **gradient descent** with

- **Update rule:** $w = w + \alpha x_i y_i$
- **Stopping criterion:** all points correctly classified or maximum number of iterations reached

During an **epoch** (iteration) we iterate over all samples of our training set, and each time we misclassify a sample, we update the weight w using the update rule defined above, then, after each **epoch**, we check the stopping criterion to decide whether to stop or start a new **epoch**.
We also need to set a **initialization value** w_{start} for our weights, and the **learning rate** α, that controls the step size of each of our updates.

1.2 The predict function

The predict function is very simple, we just use a matrix multiplication between w and the set of samples X we want to predict. We avoid absorbing the bias term in w and X, by adding the bias after doing the multiplication:

```
np.sign(self.weights[1:].transpose().dot(X.transpose()) + self.weights[0])
```

2 Preparation of training, validation and test sets

To get our 0/1 digits data set, we used the **load_digits()** function from sklearn to directly get only 0 and 1 digits already in the form of numpy arrays in this way: `X, y = load_digits(return_X_y=True, n_class=2)`. Then we used the **train_test_split()** function twice to split the data set into 60% training, 20% validation, 20% test. We also used the **stratify** parameter to have balanced classes in our datasets, and we set a random state to get reproducibility during the validation phase. We also had to convert the y (labels) from (0,1) to (-1,1) as needed for our model.

3 Validation of the model

For the validation of our model, we used the validation set (20% of our whole dataset) previously obtained using **train_test_split()**. We trained our model on our training set with different values of **initial weights** w_{start} (100 different random weights with features between 0 and 1)[2], and **learning rate** α (from 10^{-5} to 0.1) using a **grid search** approach, then we evaluated the accuracy of each trained model on our validation set. Since we are working with a very simple dataset, most of the combinations obtained an accuracy of 1. To address this issue we explored another metric for choosing our model, and, among all the models that obtained a perfect accuracy we chose the one that had the best margin γ:

$$\gamma = min_{(x_i, y_i) \in D} |w^T x_i|$$

To get a reliable measure of γ, we also had to normalize the weight vector w representing the separating hyperplane, otherwise solutions with higher norms would have been favored without actually being better.
Eventually, we obtained that the best model was the one trained with $\alpha = 0.1$ and w_{start} as defined in the weights.npy file attached, that obtained a margin $\gamma = 3.45$ on our validation set. (final weights of the model are in final_model_weights.npy)

4 Test of the model

We also tested our best configuration on a never-seen (neither in training nor in validation) test set (20% of our dataset) after retraining our model on test+validation using the previously chosen hyperparameters, and we got an accuracy of 1, with a lower margin $\gamma = 1.09$. We can say that our model generalized well (we are in a trivial case).

[1] The scalar inside the sum is always negative for a misclassified point
[2] Also here we set a seed for np.random() to obtain reproducibility

5 ChatGPT policy

We used ChatGPT to generate a first draft of our code documentation for the Perceptron.