COMP 6721 Applied Artificial Intelligence (Fall 2023)

Lab Exercise #6: Artificial Neural Networks

Question 1 Given the training instances below, use scikit-learn to implement a *Perceptron classifier*[1] that classifies students into two categories, predicting who will get an 'A' this year, based on an input feature vector $\vec{x}$. Here's the training data again:

	Feature(x)				Output f(x)
Student	'A' last year?	Black hair?	Works hard?	Drinks?	'A' this year?
X1: Richard	Yes	Yes	No	Yes	No
X2: Alan	Yes	Yes	Yes	No	Yes
X3: Alison	No	No	Yes	No	No
X4: Jeff	No	Yes	No	Yes	No
X5: Gail	Yes	No	Yes	Yes	Yes
X6: Simon	No	Yes	Yes	Yes	No

Use the following Python imports for the perceptron:

```
import numpy as np
from sklearn.linear_model import Perceptron
```

All features must be numerical for training the classifier, so you have to transform the 'Yes' and 'No' feature values to their binary representation:

```
# Dataset with binary representation of the features
dataset = np.array([[1,1,0,1,0],
                    [1,1,1,0,1],
                    [0,0,1,0,0],
                    [0,1,0,1,0],
                    [1,0,1,1,1],
                    [0,1,1,1,0],])
```

For our feature vectors, we need the first four columns:

```
X = dataset[:, 0:4]
```

and for the training labels, we use the last column from the dataset:

```
y = dataset[:, 4]
```

[1] https://scikit-learn.org/stable/modules/linear_model.html#perceptron

(a) Now, create a Perceptron classifier (same approach as in the previous labs) and train it.

(b) Let's examine our trained Perceptron in more detail. You can look at the weights it learned with:

```
print("Weights: ", perceptron_classifier.coef_)
```

And the bias, here called intercept term, with:

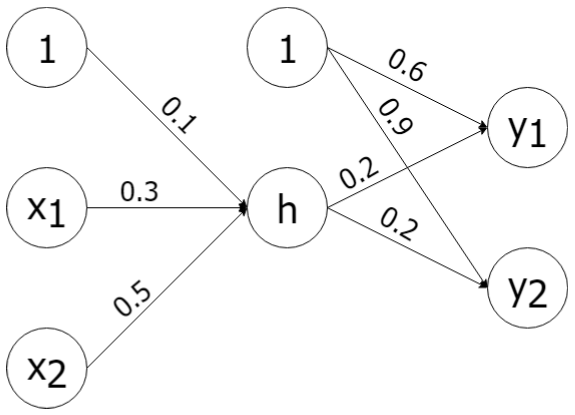
```
print("Bias: ", perceptron_classifier.intercept_)
```

The activation function is not directly exposed, but scikit-learn is using the *step* activation function. Now check how your Perceptron would classify a training sample by computing the *net* activation (input vector $\times$ weights $+$ bias) and applying the step function.

(c) Apply the trained model to all training samples and print out the prediction.

Compare the predicted labels with the actual labels from the dataset. How many predictions match the actual labels? What does this say about the performance of our classifier on the training data?

Question 2 Consider the neural network shown below. It consists of 2 input nodes, 1 hidden node, and 2 output nodes, with an additional bias at the input layer (attached to the hidden node) and a bias at the hidden layer (attached to the output nodes). All nodes in the hidden and output layers use the sigmoid activation function (σ).



(a) Calculate the output of y1 and y2 if the network is fed $\vec{x} = (1, 0)$ as input.

(b) Assume that the expected output for the input $\vec{x} = (1, 0)$ is supposed to be $\vec{t} = (0, 1)$. Calculate the updated weights after the backpropagation of the error for this sample. Assume that the learning rate $\eta = 0.1$.

Question 3 Let's see how we can build multi-layer neural networks using scikit-learn.[2]

(a) Implement the architecture from the previous question using scikit-learn and use it to learn the XOR function, which is not linearly separable.

Use the following Python imports:

```
import numpy as np
from sklearn.neural_network import MLPClassifier
```

Here is the training data for the XOR function:

```
dataset = np.array([[1,1,0],
                    [0,1,1],
                    [1,0,1],
                    [0,0,0]])
```

For our feature vectors, we need the first two columns:

```
X = dataset[:, 0:2]
```

and for the training labels, we use the last column from the dataset:

```
y = dataset[:, 2]
```

Now you can create a multi-layer Perceptron using scikit-learn's MLP (multi-layer perceptron) classifier.[3] There are a lot of parameters you can choose to define and customize, here you need to define the `hidden_layer_sizes`. For this parameter, you pass in a tuple consisting of the number of neurons you want at each layer, where the nth entry in the tuple represents the number of neurons in the nth layer of the MLP model. You also need to set the activation to 'logistic', which is the logistic Sigmoid function. The bias and weight details are implicitly defined in the function definition.

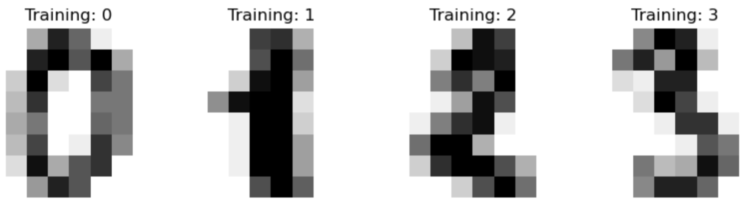
(b) Now apply the trained model to all training samples and print out its prediction.

As you see, our single hidden layer with a single neuron doesn't perform well on learning XOR. It's always a good idea to experiment with different network configurations. Try to change the number of hidden neurons to find a solution!

[2] https://scikit-learn.org/stable/modules/neural_networks_supervised.html

[3] https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

Question 4 Create a multi-layer Perceptron and use it to classify the MNIST digits dataset, containing scanned images of hand-written numerals:[4]



(a) Load MNIST from scikit-learn's builtin datasets.[5] Like before, use the `train_test_split`[6] helper function to split the digits dataset into a training and testing subset. Create a multi-layer Perceptron, like in the previous question and train the model. Pay attention to the required size of the input and output layers and experiment with different hidden layer configurations.

(b) Now run an evaluation to compute the performance of your model using scikit-learn's[7] accuracy score.

Bonus visualization: If you want to print out some example images from the test set with their predicted label, you can use the code below:

```
# Randomly select 10 images and print them with their predicted labels
n, m = 2, 5
random_indices = np.random.choice(X_test.shape[0], n*m, replace=False)
selected_images = X_test[random_indices]
selected_predictions = y_pred[random_indices]

# Plot the selected images with their predictions in a 2x5 matrix
plt.figure(figsize=(10, 4))
for i in range(n):
   for j in range(m):
      idx = i*m + j
      plt.subplot(n, m, idx + 1)
      plt.imshow(selected_images[idx].reshape((8, 8)), cmap='gray')
      plt.title(f'Predicted: {selected_predictions[idx]}')
      plt.axis('off')

plt.tight_layout()
plt.show()
```

[4] https://en.wikipedia.org/wiki/MNIST_database
[5] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html
[6] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
[7] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html

(c) In any classification task, whether binary or multi-class, it's crucial to assess how well the model is doing. Precision and recall are commonly used metrics for this purpose. For binary classification, their computation is straightforward. However, when we move to multi-class problems, the landscape becomes more complex. This is where *micro* and *macro* averaging come in, and they provide two different perspectives:

Micro-Averaging: This method gives a global view. It pools together the individual true positives, false negatives, and false positives across all classes, effectively treating the multi-class problem as a single binary classification. It provides an overall sense of how the model is performing, without differentiating between classes.

Macro-Averaging: This method breaks down the performance by class. It calculates precision and recall for each class separately and then averages them. This means every class, regardless of its size, has an equal say in the final score. It's useful for understanding the model's performance on individual classes, especially when there are imbalances in class sizes.

Both of these methods are standard in the field of machine learning and not specific to any particular library, including scikit-learn. They offer complementary perspectives: while micro-averaging might show how well the model performs overall, macro-averaging can highlight if it's struggling with any particular class.

Run an evaluation on your results and compute the precision and recall score with micro and macro averaging, using scikit-learn's `precision_score`[8] and `recall_score`.[9] Make sure you compute these on your *test* set!

(d) Use the *confusion matrix* implementation from the scikit-learn package to visualize your classification performance.

(e) K-fold cross-validation is a way to improve the training process: The data set is divided into k subsets, and the method is repeated k times. Each time, one of the k subsets is used as the test set and the other $k-1$ subsets are put together to form a training set. Then the average error across all k trials is computed. The advantage of this method is that it matters less how the data gets divided. Every data point gets to be in a test set exactly once, and gets to be in a training set $k-1$ times. The disadvantage of this method is that the training algorithm has to be rerun from scratch k times, which means it takes k times as much computation to complete an evaluation.[10]

For this task, don't use the `train_test_split` created earlier, instead use the `KFold`[11] class from the scikit-learn package to divide your dataset into

[8] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html
[9] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html
[10] https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation
[11] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

k folds. For each fold, train your MLP model on the training set and evaluate its performance on the test set. Calculate performance metrics like accuracy, precision, and recall for each fold. After all folds have been processed, compute the average performance across all folds.

Compare the average performance from cross-validation to the performance you achieved with a single train/test split.