COMP 6721 Applied Artificial Intelligence (Winter 2022)

Lab Exercise #06: 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:

```python
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:

```python
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:

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

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

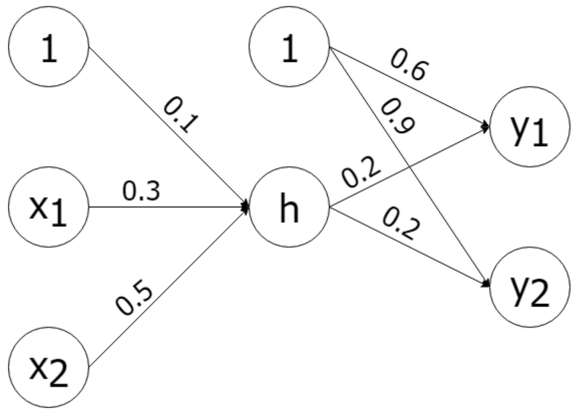
```python
y = dataset[:, 4]
```

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

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 and a bias at the hidden layer. 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 a 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:

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

Here is the training data for the XOR function:

```python
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:

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

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

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

Now you can create a multi-layer Perceptron using scikit-learn's MLP 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.

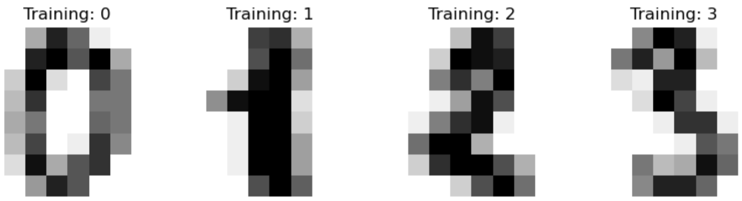
(b) Now apply the trained model to all training samples and print out its prediction.

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.

[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.

(c) In binary classification, we score the model intuitively using precision and recall metrics. But for multi-class classification, it's different: For this case, the scikit-learn package provides the implementation of precision and recall scores based on *macro* and *micro* averaging: 'micro' calculate metrics globally, by counting the total true positives, false negatives, and false positives. The 'macro' version calculates metrics for each label and finds their unweighted mean.

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]

(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

[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
[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

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.

Use `KFold`[10] from the scikit-learn package to repeat this question from the beginning, but now applying cross validation. Do you see any difference in the performance?

[10] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html