COMP 6721 Applied Artificial Intelligence (Fall 2023)

Lab Exercise #6: Artificial Neural Networks

Solutions

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:

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

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

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

Most of the solution is provided above. Here is the additional code required to create a Perceptron classifier and train it using the provided dataset:

```
perceptron_classifier = Perceptron(max_iter=40, eta0=0.1, random_state=1)
perceptron_classifier.fit(X,y)
```

The parameters we're using here are:

max_iter The maximum number of passes over the training data (aka epochs). It's set to 40, meaning the dataset will be passed 40 times to the Perceptron during training.

eta0 This is the learning rate, determining the step size during the weights update in each iteration. A value of 0.1 is chosen, which is a moderate learning rate.

random_state This ensures reproducibility of results. The classifier will produce the same output for the same input data every time it's run, aiding in debugging and comparison.

Try experimenting with these values, for example, by changing the number of iterations or learning rate. Make sure you understand the significance of setting `random_state`.

(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 × weights + bias) and applying the step function.

You can use the following code to compute the net activation on all training data samples and compare this with your results:

```
net_activation = np.dot(X, perceptron_classifier.coef_.T) +
    ↪ perceptron_classifier.intercept_
print(net_activation)
```

Remember that the step activation function classifies a sample as 1 if the net activation is non-negative and 0 otherwise. So, if a net activation is non-negative, the perceptron's step function would classify it as 1, and otherwise, it would classify it as 0.

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

This works just like for the other classifiers we used before:

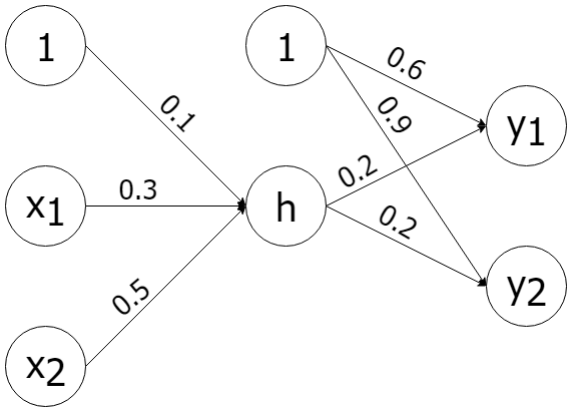
```
y_pred = perceptron_classifier.predict(X)
print(y_pred)
```

This will print the classification results like:

```
[0 1 0 0 1 0]
```

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.

$$h_{in} = b_h + w_{x_1\text{-}h}x_1 + w_{x_2\text{-}h}x_2 = (0.1) + (0.3 \times 1) + (0.5 \times 0) = 0.4$$

$$h = \sigma(h_{in}) = \sigma(0.4) = \frac{1}{1 + e^{-0.4}} = 0.599$$

$$y_{1,in} = b_{y_1} + w_{h\text{-}y_1}h = 0.6 + (0.2 \times 0.599) = 0.72$$

$$y_1 = \sigma(0.72) = \frac{1}{1 + e^{-0.72}} = 0.673$$

$$y_{2,in} = b_{y_2} + w_{h\text{-}y_2}h = 0.9 + (0.2 \times 0.599) = 1.02$$

$$y_2 = \sigma(1.22) = \frac{1}{1 + e^{-1.02}} = 0.735$$

As a result, the output is calculated as $y = (y1, y2) = (0.673, 0.735)$.

(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 $\eta = 0.1$.

$$\delta_{y_1} = y_1(1 - y_1)(y_1 - t_1) = 0.673(1 - 0.673)(0.673 - 0) = 0.148$$

$$\delta_{y_2} = y_2(1 - y_2)(y_2 - t_2) = 0.735(1 - 0.735)(0.735 - 1) = -0.052$$

$$\delta_h = h(1-h)\sum_{i=1,2} w_{h\text{-}y_i}\delta_{y_i} = 0.599(1-0.599)[0.2\times 0.148+0.2\times(-0.052)] = 0.005$$

$$\Delta w_{x_1\text{-}h} = -\eta\delta_h x_1 = -0.1\times 0.005\times 1 = -0.0005$$
$$\Delta w_{x_2\text{-}h} = -\eta\delta_h x_2 = -0.1\times 0.005\times 0 = 0$$
$$\Delta b_h = -\eta\delta_h = -0.1\times 0.005 = -0.0005$$

$$\Delta w_{h\text{-}y_1} = -\eta\delta_{y_1} h = -0.1\times 0.148\times 0.599 = -0.0088652$$
$$\Delta b_{y_1} = -\eta\delta_{y_1} = -0.1\times 0.148 = -0.0148$$

$$\Delta w_{h\text{-}y_2} = -\eta\delta_{y_2} h = -0.1\times(-0.052)\times 0.599 = 0.0031148$$
$$\Delta b_{y_2} = -\eta\delta_{y_2} = -0.1\times(-0.052) = 0.0052$$

$$w_{x_1\text{-}h,new} = w_{x_1\text{-}h} + \Delta w_{x_1\text{-}h} = 0.3 + (-0.0005) = 0.2995$$
$$w_{x_2\text{-}h,new} = w_{x_2\text{-}h} + \Delta w_{x_2\text{-}h} = 0.5 + 0 = 0.5$$
$$b_{h,new} = b_h + \Delta b_h = 0.1 + (-0.0005) = 0.0995$$

$$w_{h\text{-}y_1,new} = w_{h\text{-}y_1} + \Delta w_{h\text{-}y_1} = 0.2 + (-0.0088652) = 0.1911348$$
$$b_{y_1,new} = b_{y_1} + \Delta b_{y_1} = 0.6 + (-0.0148) = 0.5852$$

$$w_{h\text{-}y_2,new} = w_{h\text{-}y_2} + \Delta w_{h\text{-}y_2} = 0.2 + 0.0031148 = 0.2031148$$
$$b_{y_2,new} = b_{y_2} + \Delta b_{y_2} = 0.9 + 0.0052 = 0.9052$$

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.

Using the code blocks provided above, you can create the network and train it on the XOR dataset with:

```
mlp = MLPClassifier(hidden_layer_sizes=(1,),activation='logistic')
mlp.fit(X, y)
```

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

```
y_pred = mlp.predict(X)
print(y_pred)
```

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

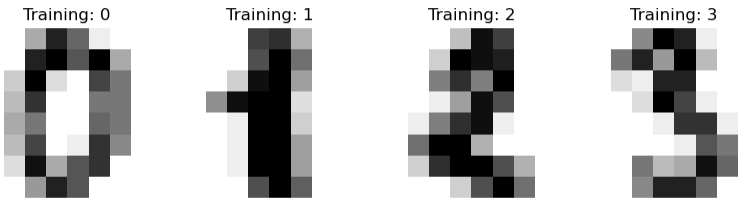
With a single hidden neuron, it can converge in theory, but it is difficult in practice, highly depending on initial weights and other hyperparameters.

With two neurons in the hidden layer, it's possible but not guaranteed to find a solution. The success of training depends on the weight initialization and the optimization algorithm's ability to find a suitable combination of weights. Often, it may get stuck in local minima.

Using three neurons in the hidden layer increases the representational capacity of the network, making it more likely to converge to a solution for the XOR problem, Try:

```
mlp = MLPClassifier(hidden_layer_sizes=(3,), activation='logistic',
    ↪ solver='lbfgs', max_iter=100000, random_state=42)
```

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.

```
import numpy as np
from sklearn import datasets
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import precision_score, recall_score
import matplotlib.pyplot as plt
```

MNIST digits is another built-in dataset in scikit-learn. First load the dataset. Since it contains two-dimensional image data, you need to flatten it, so it can be presented to our neural network as input:

```
digits = datasets.load_digits() # 2D images in feature matrix
n_samples = len(digits.images) # number of samples
data = digits.images.reshape((n_samples, -1)) # flatten 2D images into 1D
```

The third line above "flattens" the 2D image arrays, so that the resulting `data` contains a 1D-vector for each image. Thus, `data` now contains one row for each image in the dataset, with one column for each pixel in those images and its value representing a gray scale pixel in the image.

Create training and test splits (reserving 30% of the data for testing and using the rest of it for training):

[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

```
X_train, X_test, y_train, y_test = train_test_split(
   data, digits.target, test_size=0.3, shuffle=False)
```

Finally, train a neural network that can actually make predictions with:

```
mlp = MLPClassifier(hidden_layer_sizes=(100,), max_iter=1000, alpha=1e-4,
                    solver='sgd', verbose='true', random_state=1,
                    learning_rate_init=0.001)
mlp.fit(X_train, y_train)
```

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

You can evaluate the model with:

```
y_pred = mlp.predict(X_test)
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
```

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()
```

(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:

[7] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html

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!

```
pre_macro = precision_score(y_test, y_pred, average='macro')
pre_micro = precision_score(y_test, y_pred, average='micro')

recall_macro = recall_score(y_test, y_pred, average='macro')
recall_micro = recall_score(y_test, y_pred, average='micro')
```

Here, the *micro* and *macro* averages are very similar, as the classes in this dataset are mostly balanced. If one class has significantly fewer samples, macro-averaging will give you a sense of how well the model performs on that specific class compared to the others.

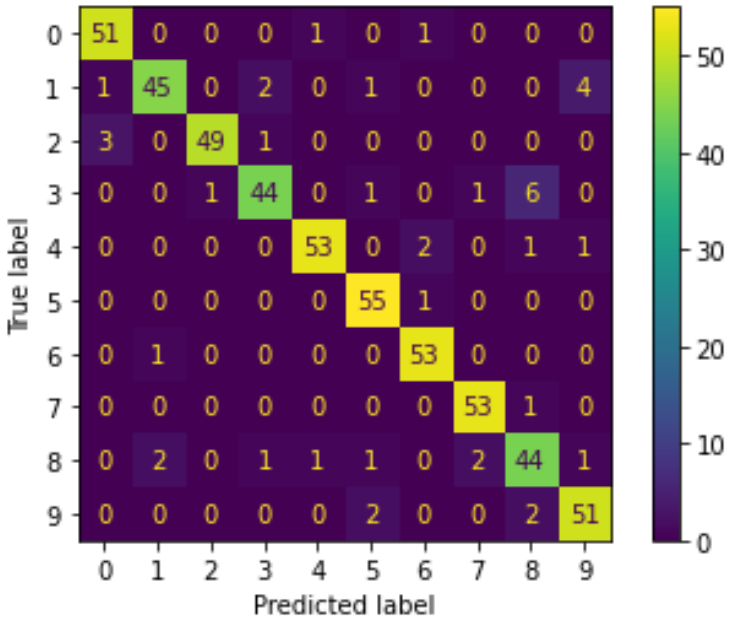
(d) Use the *confusion matrix* implementation from the scikit-learn package to visualize your classification performance.

The confusion matrix provides a more detailed breakdown of a classifier's performance, allowing you to see not just where it got things right, but where mistakes are being made. Each row in the matrix represents the true classes, while each column represents the predicted classes. It's a powerful tool to understand misclassifications, especially in multi-class problems.

```
cm = confusion_matrix(y_test, y_pred)
ConfusionMatrixDisplay(cm, digits.target_names).plot()
plt.show()
```

You should get an output similar to the following:

[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



By examining the heatmap, you can quickly identify which classes the model is confusing with others. The diagonal elements represent the number of points for which the predicted label is equal to the true label, while off-diagonal elements are those that are mislabeled by the classifier.

(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 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.

One option would be to code a loop for the number of loops, perform training and testing, and then average the results. But scikit-learn has a

[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

helper function that can do this automatically for you, `cross_val_score`,[12] here using *accuracy:*

```
from sklearn import datasets
from sklearn.neural_network import MLPClassifier

from sklearn.model_selection import KFold

digits = datasets.load_digits() # features matrix
n_samples = len(digits.images)
X = digits.images.reshape((n_samples, -1))
y = digits.target

mlp = MLPClassifier(hidden_layer_sizes=(100,), max_iter=1000, alpha=1e-4,
                    solver='sgd', verbose='true', random_state=1,
                    learning_rate_init=0.001)

# Perform 5-fold cross validation and compute accuracy scores
scores = cross_val_score(mlp, X, y, cv=10, scoring='accuracy')

print("Accuracy for each fold:")
print(scores)
print(f"Average Accuracy: {scores.mean()*100:.2f}%")
```

You can also compute multiple metrics using the `cross_validate` function:

```
scoring = ['precision_macro', 'recall_macro', 'f1_macro', 'accuracy']
scores = cross_validate(mlp, X, y, cv=5, scoring=scoring, return_train_score=False)

# Print the results from each fold
for metric, values in scores.items():
   if 'test_' in metric:
      print(f"{metric.replace('test_', '')}: {values}")

# Print the cross-fold results
for key, values in scores.items():
   print(f"{key}: {values.mean():.4f} (+/- {values.std()*2:.4f})")
```

When examining the cross-validation results, ensure you check for consistent performance across folds. Significant variability could hint at underlying dataset issues or model sensitivities. Also, while the average score offers a broad overview, individual fold results can shed light on model robustness, possibly highlighting susceptibility to certain data splits, either overfitting or underfitting.

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