Neural network classification (scikit-learn version)

Reproducing our earlier network

Previously, we wrote a simple neural network by hand to see how this works. Here we reproduce this work using `scikit-learn`, a basic machine learning package.

You should already have `numpy` installed. You may need to install `scikit-learn` on your computer. Do this using the following command. Ask for help if you need it.

```
python -m pip install --user scikit-learn
```

First, let's import some packages we are going to use.

```
import matplotlib.pyplot as plt
import numpy as np
from random import random
from sklearn.neural_network import MLPClassifier
```

Please download the training data (or reuse the version you already downloaded) and save it in the same location as a new Python file. Here we import the data and plot it, exactly as we did in the previous program.

```
f = open("training-data.csv", "r")

data = []
labels = []

for line in f:
    line = line.rstrip('\r\n')
    parts = line.split(",")
    data.append((float(parts[0]),float(parts[1])))
    labels.append(float(parts[2]))

f.close()

group0_x1 = []
group0_x2 = []
```

```
group1_x1 = []
group1_x2 = []

for i in range(len(data)):
    if labels[i] == 0: # group 0
        group0_x1.append(data[i][0])
        group0_x2.append(data[i][1])
    else: # group 1
        group1_x1.append(data[i][0])
        group1_x2.append(data[i][1])

plt.scatter(group0_x1,group0_x2,c="blue",marker="+",label="Group 0")
plt.scatter(group1_x1,group1_x2,c="purple",marker="+",label="Group 1")
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Training data")
plt.legend()
plt.show()
```

scikit-learn wants the input data to be in the form of `numpy` arrays. These are a data structure provided by the `numpy` package, and we can convert a list into an array using `np.array()`.

```
train_data = np.array(data)
train_labels = np.array(labels)
```

We create our model. `MLPClassifier` is a Multi-layer Perceptron classifier, the type of neural network we are going to use. A perceptron is the type of neuron we previously created by hand. In a neural network the first layer of neurons is called the input layer, and the last is called the output layer. The ones in between are called the hidden layers - hidden in that they neither interact directly with the input or output. Here we ask for no hidden layers, to match what we previously did manually.

```
model = MLPClassifier(hidden_layer_sizes=())
```

Now we have created a neural network, we can proceed to train it. Last time, we wrote a function as the activation function and made a loop to adjust the weights and bias based on the training data. Here scikit-learn will do all this for us, the command we need is `.fit()`, to which we pass the training data and labels.

```
model.fit(train_data, train_labels)
```

That's it! Now we can test our trained network to see how it performs. This is again similar to the approach we took to testing when we did this manually, except that the input coordinates must be a `numpy` array and the model does the prediction for us without us having to code up what the neurons are doing.

```
group0_x1 = []
group0_x2 = []
group1_x1 = []
group1_x2 = []
```

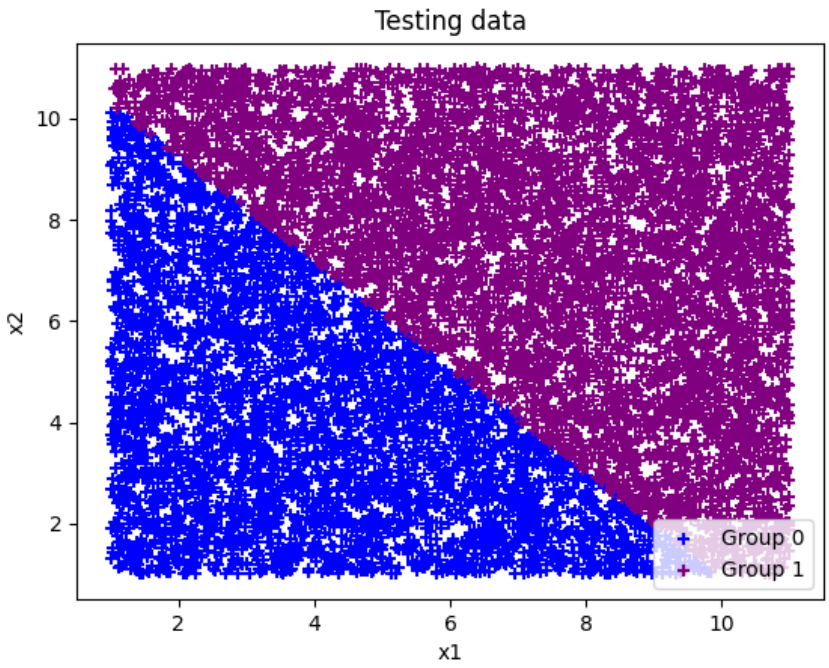
```
for i in range(0,10**4):
    this_test = np.array([[random()*10+1,random()*10+1]])

    prediction = model.predict(this_test)[0]

    if prediction == 0:
        group0_x1.append(this_test[0][0])
        group0_x2.append(this_test[0][1])
    else:
        group1_x1.append(this_test[0][0])
        group1_x2.append(this_test[0][1])
```

We stored the testing coordinates in either `group0_x1` and `group0_x2` or `group1_x1` and `group1_x2`, depending on whether the model predicted these coordinates were in group 0 or group 1. We can plot the results.

```
plt.scatter(group0_x1,group0_x2,c="blue",marker="+",label="Group 0")
plt.scatter(group1_x1,group1_x2,c="purple",marker="+",label="Group 1")
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Testing data")
plt.legend()
plt.show()
```

Performance

How did the network perform? You may have got a nice clear delineation between two groups, like this.



However, if you run this multiple times, you may get something that performs less well - and you may have got mixed performance with the network we coded by hand as well.

The reason for this is that our network is not very sophisticated. `MLPClassifier` trains with 200 epochs (the one we coded by hand did two!), but even so it contains only the input and output layers. The use of more neurons arranged into hidden layers allows for more complicated interactions between the inputs and the weights, meaning the network is capable of capturing more nuanced information. In general, the more hidden layers the more sophisticated, but also more means it will be slower to run.

Hidden layers

When we set up our model, we told `MLPClassifier` to use no hidden layers by passing an empty tuple to `hidden_layer_sizes`. What this expects is a tuple listing the numbers of neurons in each layer. For example

```
model = MLPClassifier(hidden_layer_sizes=(10,5))
```

asks for a neural network with two hidden layers, the first containing ten neurons and the second containing five.

Try editing your code to include some hidden layers like this and you should find the performance is more reliable.

A more sophisticated example

The training data we generated for the example above was very distinct: values around (3,3) are in group 0 and values around (9,9) are in group 1, with no overlap between the groups. This is because it was generated for a network of one neuron that couldn't deal with any nuance.

Now we have a way to train more sophisticated networks, we can assign them more complicated tasks.

Please download training-data2.csv and use the code below to plot this. You should see a blob of data in group 0 with a ring of data from group 1 around it.

```
import matplotlib.pyplot as plt

f = open("training-data2.csv", "r")

data = []
labels = []

for line in f:
    line = line.rstrip('\r\n')
    parts = line.split(",")
    data.append((float(parts[0]),float(parts[1])))
    labels.append(float(parts[2]))

f.close()

group0_x1 = []
group0_x2 = []
group1_x1 = []
group1_x2 = []

for i in range(len(data)):
    if labels[i] == 0: # group 0
        group0_x1.append(data[i][0])
        group0_x2.append(data[i][1])
    else: # group 1
        group1_x1.append(data[i][0])
        group1_x2.append(data[i][1])

plt.scatter(group0_x1,group0_x2,c="blue",marker="+",label="Group 0")
plt.scatter(group1_x1,group1_x2,c="purple",marker="+",label="Group 1")
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Training data")
plt.legend()
plt.show()
```

Classifying points in this dataset is a more sophisticated task because there are points of both groups that have x_1 coordinates around 50 and points of both groups that have x_2 coordinates around 50. It is only points that have both coordinates around 50 that are in group 0.

A Multi-layer Perceptron classifier is sensitive to scaling, so it is recommended to scale the data to a standard range before giving it to the model. scikit-learn provides a function to do this: `StandardScaler`. Here we use this to scale the training data to have mean `0` and variance `1`.

```
from sklearn.preprocessing import StandardScaler

train_data = np.array(data)
train_labels = np.array(labels)

scaler = StandardScaler()
scaler.fit(train_data)
train_data = scaler.transform(train_data)
```

Now we can train the model in the same way as above, with some hidden layers.

```
model = MLPClassifier(hidden_layer_sizes=(10,5))
model.fit(train_data, train_labels)
```

Having trained the model, we can produce some testing data. Here we create two slightly wider rings with the same centres as above - slightly wider because we hope the trained model is capable of dealing with data that aren't quite the same as the inputs. We store the coordinates as a `numpy` array, and we must scale the testing data in the same way as we did the training data. After this, we put the unscaled coordinates in lists according to which group the model predicts they belong to, as previously.

```
def gen_coords(centre,r):
    theta = random()*math.pi/2 # random angle
    distance = random()*(r[1]-r[0])+r[0] # random distance from centre
    # equally likely quadrant
    xfactor = (-1)**randint(0,1)
    yfactor = (-1)**randint(0,1)
    x = centre[0] + xfactor*distance*math.cos(theta)
    y = centre[1] + yfactor*distance*math.sin(theta)
    return (x,y)

# slightly bigger rings
centre0 = (50,50)
r0 = (0,20)
centre1 = (50,50)
r1 = (70,110)

for i in range(0,10**4):
    if random()<0.5:
        xy = gen_coords(centre0,r0)
    else:
        xy = gen_coords(centre1,r1)

    this_test = np.array([[xy[0],xy[1]]])
    this_test = scaler.transform(this_test)

    prediction = model.predict(this_test)[0]

    if prediction == 0:
        group0_x1.append(xy[0])
        group0_x2.append(xy[1])
    else:
```

```
        group1_x1.append(xy[0])
        group1_x2.append(xy[1])
```

Now we can use the same code as before to plot the testing data and see how well the model has performed.

```
plt.scatter(group0_x1,group0_x2,c="blue",marker="+",label="Group 0")
plt.scatter(group1_x1,group1_x2,c="purple",marker="+",label="Group 1")
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Testing data")
plt.legend()
plt.show()
```