Simple_neural_network_(1_neuron)

February 1, 2023

1 1. Importing necessary packages

`numpy` to handle matrices, `matplotlib.pyplot` fo illustrate graphics and `accuracy_scor` to measure the accuracy of the model. Then `sklean.datasets` to generate random data for simulating a dataset, more precisely we use `make_blobs` that actually generate isotropic Gaussian blobs for clustering.

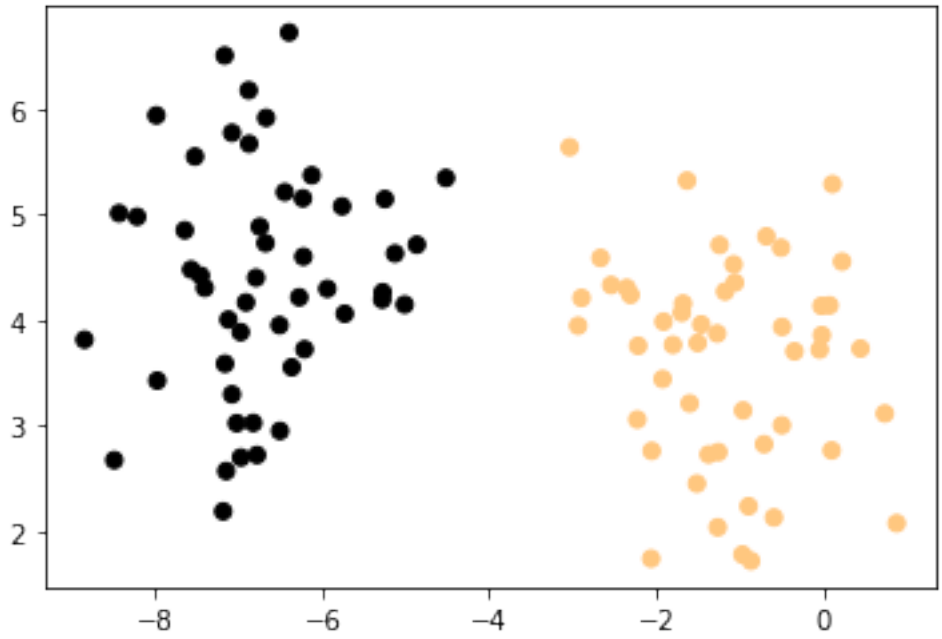
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_blobs
```

2 2. Simulating a Dataset

X represents the input data, and it is a $n \times 2$ matrix which contains n observations of pairs $x_i = (x_{1,i}, x_{2,i}) \in \mathbb{R}^2$. $y_i \in \{0; 1\}$ is the response variable which is binary and Y is the output vector which contains the n observed responses y_i's.

[2]:
```
X, y = make_blobs(n_samples=100, n_features=2, center_box=(-8.0, 8.0),␣
↪centers=2, random_state=7)
y = y.reshape((y.shape[0], 1))

print('dimensions de X:', X.shape)
print('dimensions de y:', y.shape)

plt.scatter(X[:,0], X[:, 1], c=y, cmap="copper")
plt.show()
```

```
dimensions de X: (100, 2)
dimensions de y: (100, 1)
```



Next, we will split the artificial neuron into several pieces, where each piece is defined as a function as follows:

3 3. Defining model functions

3.1 3.1. initialization

Random values are assigned to the parameters to start the first forward propagation before starting the backward ster that consists on updating their values.

[3]:
```
def initialization(X):
    W = np.random.randn(X.shape[1], 1) # generating a samples from the␣
 ↪"standard normal" distribution to
    b = np.random.randn(1)             # as weight matrix W and bias b
    return (W, b)
```

3.2 3.2 Sigmoid function (logit)

Z is the linear combination of the values of X with the coefficients W plus a bias term. And A represents the value of the sigmoid function at this point Z.

[4]:
```
def model(X, W, b):
    Z = X.dot(W) + b                   # Z is a linear combination of X and W␣
 ↪plus a bias term
    A = 1 / (1 + np.exp(-Z))           # A is the value of the sigmoid/logit␣
 ↪function applied to Z
```

```
    return A
```

3.3 3.3 Calculating the loss function : log-likelihood

We can express the likelihood as follows !

$$L = \prod_{i=1}^{n} a_i^{y_i} \times (1 - a_i)^{1-y_i}$$

instead of maximising the log-likelihood, we prefer to minimise its negative version putting a minus sign in front of it and considering it as a loss function to minimise.

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^{n} y_i \log(a_i) + (1 - y_i) \log(1 - a_i)$$

[5]:
```
def log_loss(A, y):
    return 1 / len(y) * np.sum(-y * np.log(A) - (1 - y) * np.log(1 - A))  #␣
↪log-likelihood
```

3.4 3.4 Obtaining the gradient values and updating

Using the sigmoid as an activation function leads to the expression below, they might change if one uses a different activation function.

[6]:
```
def gradients(A, X, y):
    dW = 1 / len(y) * np.dot(X.T, A - y)        # Gradiant calculus to update␣
↪weights
    db = 1 / len(y) * np.sum(A - y)             #   ...       ...    to update␣
↪bias
    return (dW, db)
```

[7]:
```
def update(dW, db, W, b, learning_rate):
    """updating function"""
    W = W - learning_rate * dW
    b = b - learning_rate * db
    return (W, b)
```

The function that calculate the value of sigmoid at the poing X with the actual values of parameters W and b, and then returns a binary value, 1 if the value exceeds some threshold and 0 if not. We fixed by default the value of the threshold to 0.5.

[8]:
```
def predict(X, W, b, border=0.5):
    """Choosing arbitrary the border to be at 0.5. \n
    It might be better to consider the border as a hyperparameter to␣
↪calibrate"""
    A = model(X, W, b)
    # print(A)
    return A >= border
```

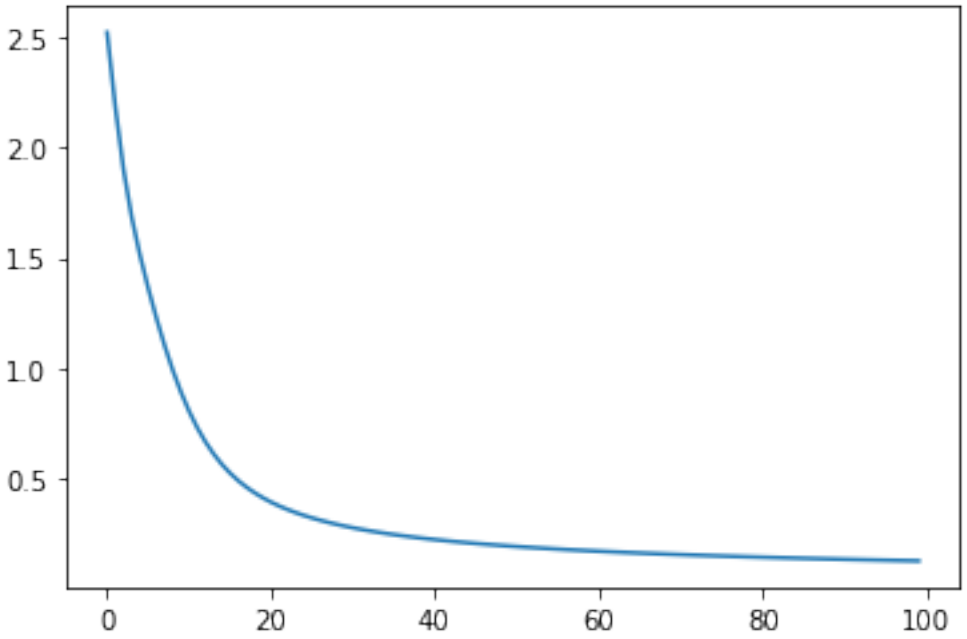
4 4. The model

We finally defined all needed functions. We're now able to put all that pieces inside one function called `artificial_single_neuron` that takes X and y as entries and some additional parameters as the learning_rate and the number of iterations, and after running, it returns the updated values of parameters minimising the loss function.

[9]:
```
def artificial_single_neuron(X, y, learning_rate = 0.1, n_iter = 100):
    # initialisation W, b
    W, b = initialization(X)

    Loss = []

    for i in range(n_iter):
        A = model(X, W, b)
        Loss.append(log_loss(A, y))
        dW, db = gradients(A, X, y)
        W, b = update(dW, db, W, b, learning_rate)

    y_pred = predict(X, W, b)
    print(accuracy_score(y, y_pred))

    plt.plot(Loss)
    plt.show()

    return (W, b)
```

[10]:
```
W, b = artificial_single_neuron(X, y)
```

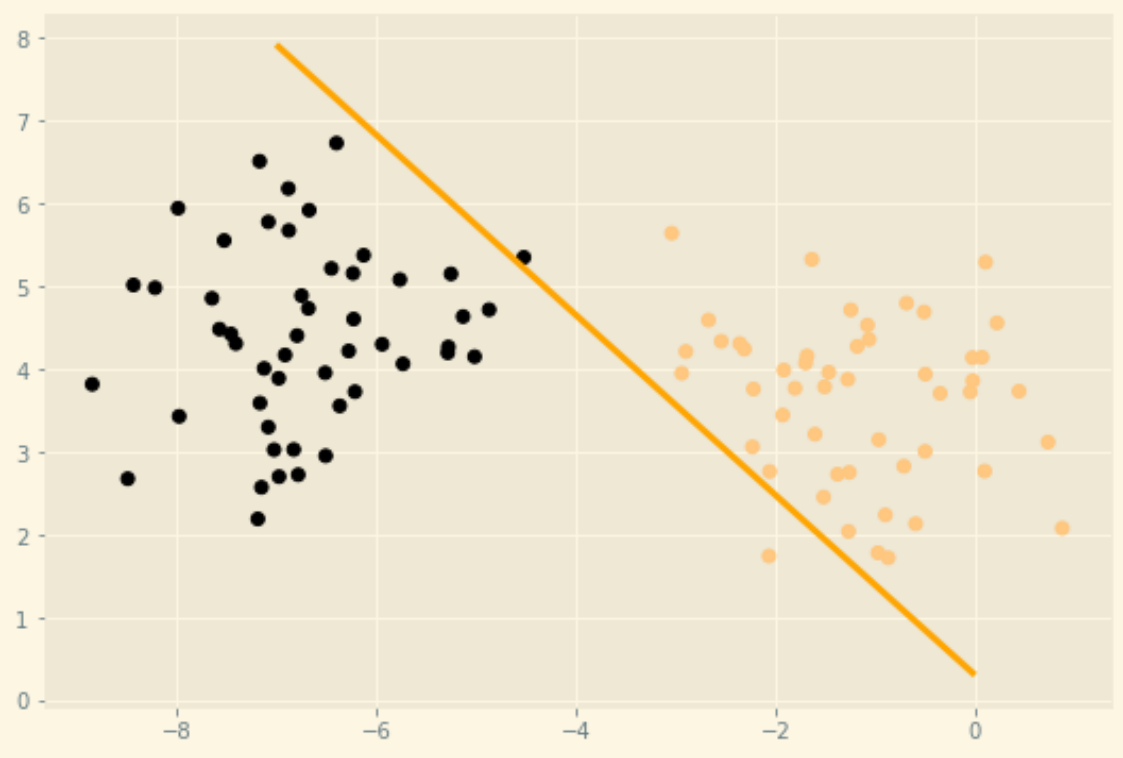
```
0.98
```



5 3. Frontiere de décision

[11]:
```
plt.style.use('Solarize_Light2')
```

[12]:
```
fig, ax = plt.subplots(figsize=(9, 6))

ax.scatter(X[:,0], X[:, 1], c=y, cmap='copper')

x1 = np.linspace(-7, 0, 100)
x2 = ( - W[0] * x1 - b) / W[1]

ax.plot(x1, x2, c='orange', lw=3)
plt.show()
```

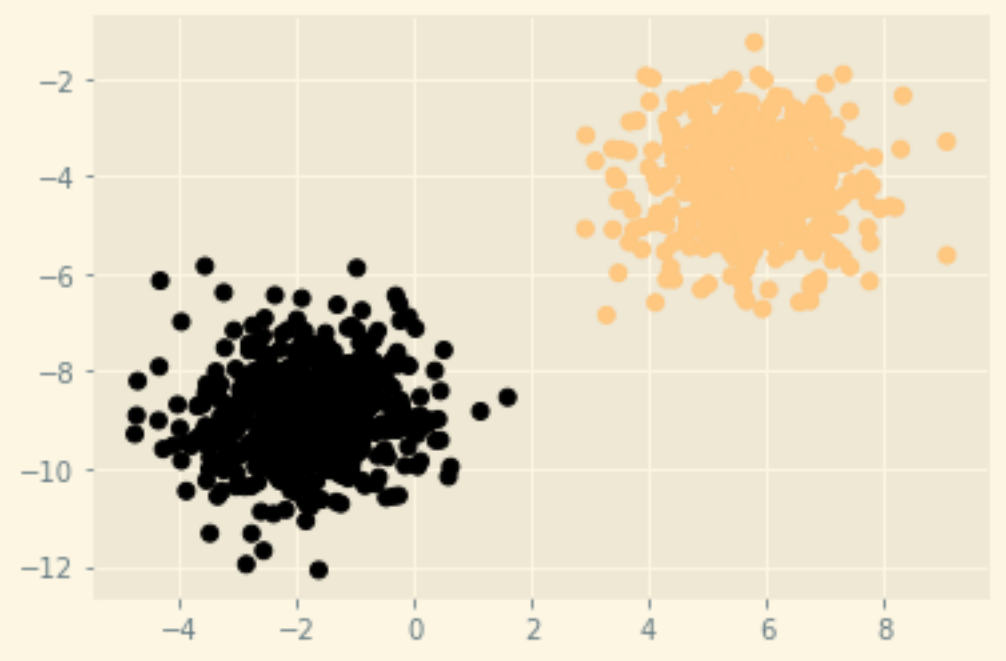


6 5. Additional examples

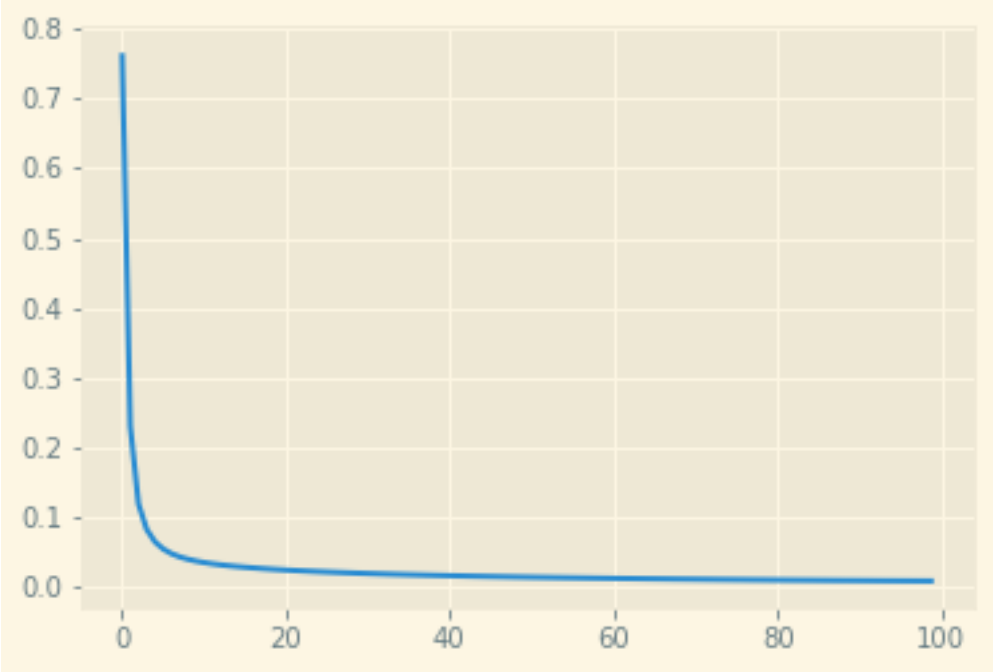
[13]:
```
X, y = make_blobs(n_samples=1000, n_features=2, centers=2, random_state=40)
y = y.reshape((y.shape[0], 1))

print('dimensions de X:', X.shape)
print('dimensions de y:', y.shape)

plt.scatter(X[:,0], X[:, 1], c=y, cmap="copper")
plt.show()

W, b = artificial_single_neuron(X, y)


fig, ax = plt.subplots(figsize=(9, 6))

#plt.style.use('dark_background')
# plt.style.use('Solarize_Light2')

# fig.suptitle("Graphic")
ax.scatter(X[:,0], X[:, 1], c=y, cmap="copper")
```

```
x1 = np.linspace(-1, 5, 100)
x2 = ( - W[0] * x1 - b) / W[1]

ax.plot(x1, x2, c='red', lw=3)
plt.show()
```

```
dimensions de X: (1000, 2)
dimensions de y: (1000, 1)
```




```
1.0
```


0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
20
40
60
80
100


5
0
−5
−10
−15
−4
−2
0
2
4
6
8

7 Conclusion

Only with few lines of code, we've been able to build a program that processes a backward-forward propagation, updating parameters using data and gradient descent, and then output the optimal weights that minimise the loss function the negative version of likelihood.