Single Cell Perceptron in Python: Implementation and Results

Irfan S
Dept. of Linguitics / UIUC
irfans2@illinois.edu

1 Data

We will be using five datasets to train and observe the model. The first three are the truth tables generated by the boolean functions 'AND', 'OR', and 'XOR' on three input variables. We expect our model to learn the 'AND' and 'OR' functions, but not 'XOR'.

The final two datasets are derived from the Iris flower dataset (Lichman, 2013) . The original dataset has three output classes, one of which ('iris-setosa') is linearly separable from the other two ('iris-versicolor' and 'iris-virginica'), which cannot be linearly separated. On this basis, the dataset is divided into two derived sets with binary class outputs, corresponding to the pairs of linearly and non-linearly variables respectively. The perceptron model is expected to converge only on the linearly separable data.

2 Model

The model implementation follows the description of the perceptron learning algorithm in Ch.4 of Rojas (1996). The model is initialised with randomised weights corresponding to the number of input variables. A bias value is added to the weight vector, and the input vector extended by 1 to allow learning of this parameter.

The activation function of the model is defined as the dot product of the input and weight vectors. The model predicts a positive class if the activation function returns a positive value, and a negative class otherwise. During training, the error for each row of inputs is calculated as the difference of the true output and the predicted output. For each row of input, x, the weight vector w is adjusted by an amount $learning - constant * error * x$. Training is continued until either the model converges or the specified number of iterations are completed. The weights corresponding to the lowest error on the training data are stored.

Python code for the implementation is provided in Listing 1.

3 Results

Now we present graphs of the model's error rate while training on the different datasets.

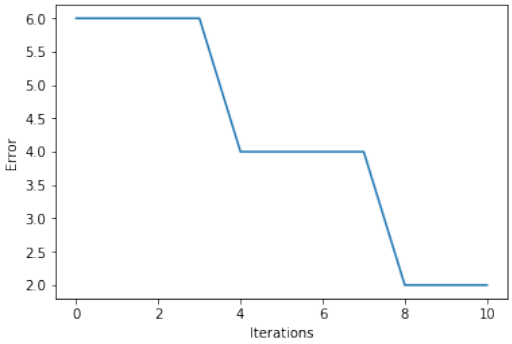

Figure 1: Learning the AND function

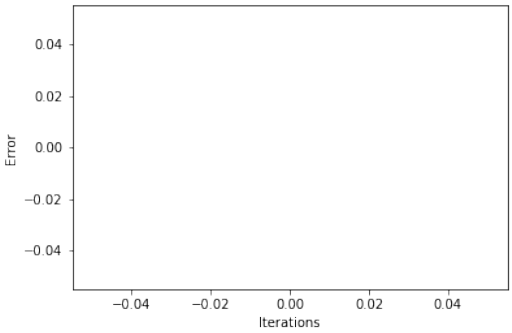

Figure 2: Learning the OR function

We can see through these plots that our perceptron model is able to quickly converge on the three linearly separable datasets, namely the AND/OR function tables, and the linearly separable classes of the Iris data (Figures 1, 2 and 4). However, the

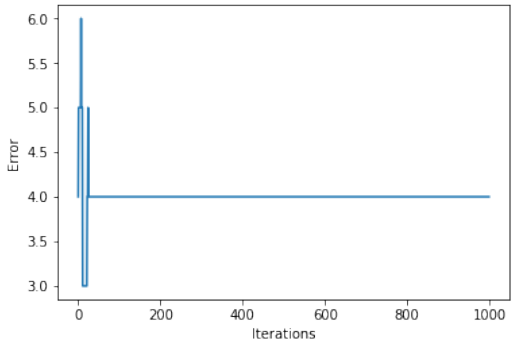

Figure 3: Learning the XOR function

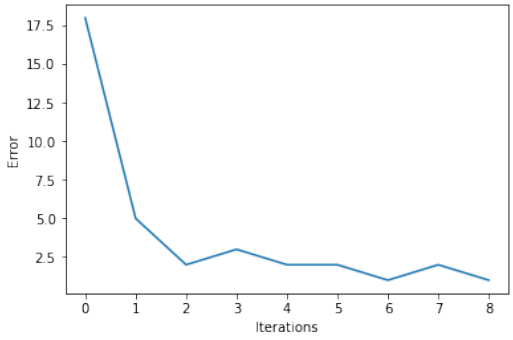

Figure 4: Training on the Iris 'linear' dataset

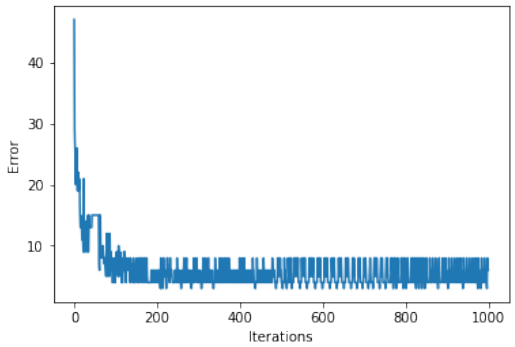

Figure 5: Training on the Iris 'non-linear' data

model fails to train successfully on the XOR function data or the non-linearly separable classes of the Iris data (Figures 3 and 5) even after training for 1000 iterations. Clearly, a simple perceptron model is only able to function as a linear classifier.

References

M. Lichman. 2013. UCI machine learning repository. http://archive.ics.uci.edu/ml.

Raúl Rojas. 1996. *Neural Networks*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-61068-4.

```
import random
import numpy as np
import sys


class Perceptron:
    def __init__(self, data, weights = None):
        random.shuffle(data)
        inputs = np.array([[float(x) for x in row[0:-1]] for row in data])
        self.inputs = np.hstack((inputs, [[1]] * len(inputs))) # Append 1 to each input row, for the bias weight
        self.outputs = np.array([float(row[-1]) for row in data])
        self.numInputs = len(self.inputs[0])
        if weights == None:
            weights = np.array([random.uniform(0, 100) \
                                    for x in range(self.numInputs)])
            weights[-1] = -1 # set initial value of bias weight
        self.weights = weights
        self.error = float(sys.maxsize) # initialise error to some very high value
        self.smallestError = self.error
        self.bestWeights = self.weights
        self.fitHistory = []

    def predict(self, x_i):
        y = np.dot(x_i, self.weights) # Activation function is the dot product of input vector and weight vector
        return 1 if y > 0 else 0

    def fit(self, lr=1, numIters = 100, breakSoon=True):
        errorList = []
        for iter in range(numIters):
            totalError = 0.0
            for i in range(len(self.outputs)):
                pred = self.predict(self.inputs[i])
                error = self.outputs[i] - pred # Error is the difference between true and predicted class
                self.weights = self.weights + \
                               lr * error * self.inputs[i] # multiplying with the error yields a positive or negative adjustm
                totalError += abs(error)

            self.saveBestFit(self.weights, totalError)
            if breakSoon:
                if totalError == 0.0:
                    break
            self.printWeights()
            errorList.append(totalError)

        self.fitHistory = errorList # Store error history for convenient plotting
        self.error = totalError

    def saveBestFit(self, w, e): # Store the best performing weights for reuse
        if e < self.smallestError:
            self.smallestError = e
            self.bestWeights = w

    def printWeights(self):
        print("\t".join(map(str, self.weights)), file=sys.stderr)

    def test(self): # Ideally we should split data into train/test sets to feed this method. For now, just use the data passe
        e = 0.0
        for i in range(len(self.inputs)):
            pred = self.predict(self.inputs[i])
            e += self.outputs[i] - pred
        print(e, file=sys.stdout)

    def __str__(self):
        s = "inputs (1 sample): {}\n".format(self.inputs[0])
        s += "weights: {}\n".format(self.weights)
        s += "error: {}\n".format(self.error)
        return s
```

Listing 1: Model Implementation