NOTES ON
PERCEPTRONS

Based On

Udacity's Deep Learning Nanodegree

By

0xLeo (github.com/0xleo)

March 13, 2019

DRAFT X.YY
Missing: . . .

Contents

1 The Perceptrons **2**
1.1 What is perceptron? 2
1.2 Its architecture 2
1.3 Perceptron for logical operations 3
1.3.1 AND perceptron 3
1.3.2 OR perceptron 4
1.3.3 NOT perceptron 4
1.3.4 XOR perceptron 5
1.3.5 Linear separability 5
1.4 The perceptron trick - intuition 5
1.5 Perceptron weight correction - the algorithm 7
1.6 Algorithm implementation 8

A Appendices **9**
A.1 Proof of Cor. 1.2 (perceptron boundary line) 10
A.2 Perceptron with online training in Python. 11

1 The Perceptrons

1.1 What is perceptron?

DEFINITION 1.1. *Perceptron is an algorithm for binary classification that uses a linear prediction function.*

So if that input data are $\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^n$, each with features $\begin{bmatrix} x_1 & x_2 & \dots & x_d \end{bmatrix}$ then a perception separates them in two classes in the feature space – "positive" and "negative" instances.

For convenience, unless otherwise stated, we'll consider them 2D, i.e. the features are x_1, x_2. In a 2D fearure space, a perceptron draws a boundary line to separate the data in two classes - one above the line and one below. More generally, For a d-dimensional input space, the decision boundary is an $d-1$-dimensional hyperplane.

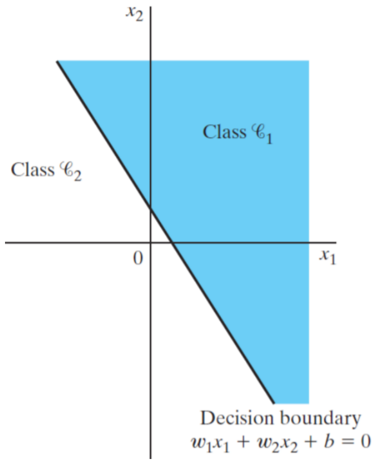


Fig. 1. The decision boundary (line) separates the x_1, x_2 space in two semi-planes.

1.2 Its architecture

The perceptron takes as input a bunch of data $\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^n$ and outputs a label value $y \in \{0, 1\}$ for each, which indicates the predicted class where the data sample belongs. Let's analyse the architecture in 2D.

The inputs are x_1, x_2. x_1 is weighted (multiplied) by some value x_1 and x_2 by w_2. The perceptron also needs some bias so that the decision boundary doesn't always cross the origin. This can be considered as an additional input $x_0 = 1$, multiplied by weight w_0. Their weighted sum $S = x_1w_1 + x_2w_2 + b$ is evaluated and passed through an activation function f, whose value determines the input's class, so the final predicted output is:

$$y_{pred} = f(x_1w_1 + x_2w_2 + b) \tag{1.1}$$

Using the dot product, this can be written more compactly as:

$$y_{pred} = f(\mathbf{w} \cdot \mathbf{x} + b) \tag{1.2}$$

$\mathbf{w}$ is the weight vector and b (a scalar) the bias. A simple activation function f is the step function.

DEFINITION 1.2. *The step function is defined as*

$$f(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases} \tag{1.3}$$

So for our model the predicted output $f(\mathbf{w} \cdot \mathbf{x} + b) \in \{0, 1\}$ depends simply on the sign of $\mathbf{w} \cdot \mathbf{x} + b$. In 2D, for positive instances we have $w_1x_1 + w_2x_2 + b \geq 0$ and for negative ones $w_1x_1 + w_2x_2 + b < 0$.

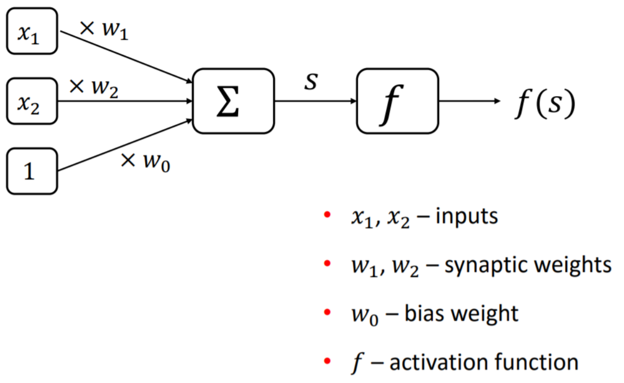


Fig. 2. Perceptron model.

So the goal of the perceptron is for each data point $\mathbf{x} \in \mathbb{R}^n$ find some $\mathbf{w} \in \mathbb{R}^n$ and an offset $b \in \mathbb{R}$ such that it can correctly decide whether $\mathbf{w} \cdot \mathbf{x} + b \geq 0$ or $\mathbf{w} \cdot \mathbf{x} + b < 0$. Therefore if the number of features of the input is n (e.g. 2 in the 2D space) then the number of coefficients to be learned is $n+1$ (e.g. $w_0,\ w_1,\ b$ for the 2D space). The decision boundary is a line for 2D inputs, a plane for 3D, etc., and in general it's called hyperplane.

Definition 1.3. *The equation of the decision boundary hyperplane is*

$$\boldsymbol{w} \cdot \boldsymbol{x} + b = 0 \tag{1.4}$$

or equivalently

$$\sum_{i=1}^{n} w_i x_i + b = 0 \tag{1.5}$$

w_i and b are to be learned.

1.3 Perceptron for logical operations

1.3.1 AND perceptron

The perceptron can be used to classify samples $\mathbf{x} = (x_1, x_2)$ for which the output is defined by the AND operation, as given by the following truth table.

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

We therefore want for the predicted output:

$$f(0,0) = 0 \Rightarrow 0 \cdot w_1 + 0 \cdot w_2 + b < 0 \tag{1}$$

$$f(0,1) = 0 \Rightarrow 0 \cdot w_1 + 1 \cdot w_2 + b < 0 \tag{2}$$

$$f(1,0) = 0 \Rightarrow 1 \cdot w_1 + 0 \cdot w_2 + b < 0 \tag{3}$$

$$f(1,1) = 1 \Rightarrow 1 \cdot w_1 + 1 \cdot w_2 + b > 0 \tag{4}$$

A set of weights that satisfies Eq. (1)-(4) is $w_1 = 1,\ w_2 = 1,\ b = -1.3$. Therefore in 2D space the inputs would be mapped in two classes as follows.

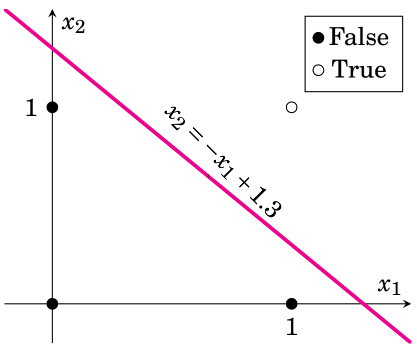


Fig. 3. Boundary line for prediction $y = x_1$ AND x_2.

A good boundary line that separates the data in two classes is $x_2 = -x_1 + 1.3$ (the weights and offset were found from Eq. (1)-(4)). The semi-plane for which the prediction is TRUE, i.e. $y = 1$, is:

$$x_1 + x_2 - 1.3 \geq 0 \Rightarrow$$

$$1 \cdot x_1 + 1 \cdot x_2 + (-1.3) = 0$$

Therefore the data can be classified by an perceptron with $w_1 = 1,\ w_2 = 1,\ b = -1.3$.

1.3.2 OR perceptron

Similarly, for the OR truth table and boundary line we have:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

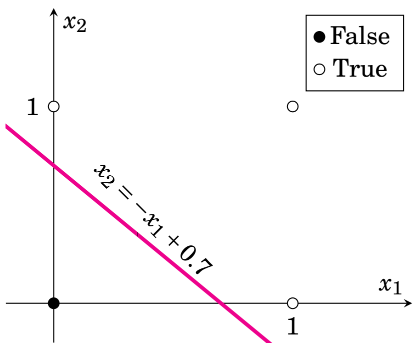


Fig. 4. Boundary line for prediction $y = x_1$ OR x_2.

For the prediction $y = 1$ (TRUE) we have:

$$x_1 + x_2 - 0.7 \geq 0 \Rightarrow$$

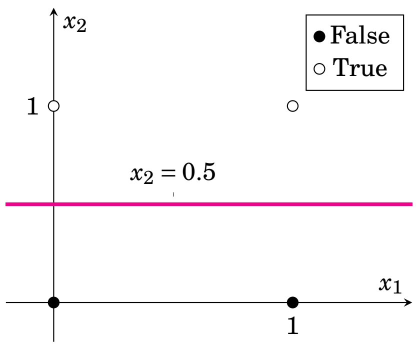
$$1 \cdot x_1 + 1 \cdot x_2 + 0.7 = 0$$

Therefore the perceptron weights are $w_1 = 1,\ w_2 = 1,\ b = -0.7$.

1.3.3 NOT perceptron

The NOT operation is slightly different from the other two as it's defined for only one input and doesn't care about the other. Let's try to design a perceptron for $y = NOT(x_2)$. In the following truth table, X stands for "don't care".

x_1	x_2	y
X	0	1
X	1	0



1.3.4 XOR perceptron

For the XOR operation, the truth table and output is:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

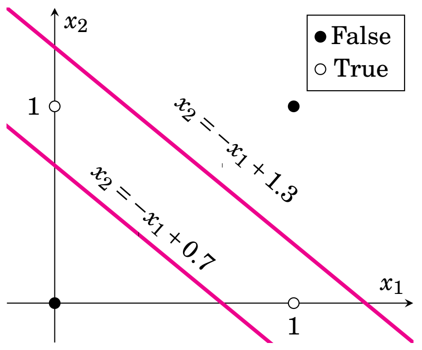


Fig. 6. XOR outputs and the two boundary lines needed to classify them.

However, the is *no single boundary line* that can correctly separate the data. Two lines are needed as shown in 6. The area we want to classify as $y = 1$ is desribed by:

$$x_1 + x_2 - 0.7 \geq 0 \text{ and } x_1 + x_2 - 1.3 < 0 \Rightarrow$$

$$(x_1\, OR\; x_2)\ AND\ \ (NOT\ \ (x_1\, AND\; x_2))$$

Schematically, the XOR perceptron is described as follows:

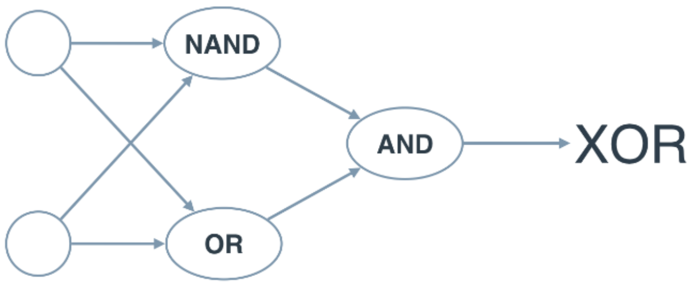


Fig. 7. XOR design as a combination of other perceptrons.

1.3.5 Linear separability

Linear separability means that the two input classes X_0, X_1 can be perfectly separated by a straight line. Mathematically, this is expressed as:

DEFINITION 1.4. *Let X_0, X_1 be two disjoint sets of the n-dimensional input data X. Then the sets are linearly separable if there exist $n+1$ real numbers $w_0, w_1, \ldots, w_{n-1}, b$ such that:*

$$\sum_{i=1}^{n} w_i x_i - b > 0 \quad \forall x \in X_0 \quad \text{and} \tag{1.6}$$

$$\sum_{i=1}^{n} w_i x_i - b < 0 \quad \forall x \in X_1 \tag{1.7}$$

1.4 The perceptron trick - intuition

When a point is classified there are 3 possibilities:

1. It is correctly classified, i.e. $y = y_{pred}$.
2. A positive sample is misclassified, i.e. $y = 1$ and $y_{pred} = 0$.

3. A negative sample is misclassified, i.e. $y = 0$ and $y_{pred} = 1$.

For every misclassfied point, the weights w_1, w_2 and bias b need to be corrected. Before describing the perceptron correction trick, the maths of the hyperplane needs to be understood. For simplicity, we'll consider the hyperplane as a line but the observations below can be generalised.

COROLLARY 1.1. *For the boundary line $w_1x_1 + w_2x_2 + b = 0$, The vector $\boldsymbol{w} = (w_1, w_2)$ is perpendicular to it.*

COROLLARY 1.2. *For the boundary line $w_1x_1 + w_2x_2 + b = 0$, its distance from the origin is $\frac{b}{||\boldsymbol{w}||}$.*

Proof.
See A.1. □

Denoting negative instances as $y = 0$ and positive as $y = 1$, the perceptron update trick is based on the following heuristic that handles false cases.

- $y = 1$ but the perceptron thinks that $y_{pred} = 0 \Rightarrow \mathbf{w}_{old}\mathbf{x} + b_{old} < 0$.

  Then the updates should be

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \mathbf{x}$$
$$b_{new} = b_{old} + 1$$

  Indeed, in this case for the output of the new updates we have

$$\begin{aligned} \mathbf{w}_{new} \cdot \mathbf{x} + b_{new} &= (\mathbf{w}_{old} + \mathbf{x})\mathbf{x} + b_{old} + 1 \\ &= \mathbf{w}_{old} \cdot \mathbf{x} + b_{old} + \mathbf{x} \cdot \mathbf{x} + 1 \\ &> \mathbf{w}_{old} \cdot \mathbf{x} + b_{old} \end{aligned}$$

  Therefore the updated sum is more positive than the old, hence has better chance of being classified correctly by the step function. Geometrically, this effect is illustated for the line below, for which $b = 0$ for simplicity so the effect on the offset is not demonstrated.

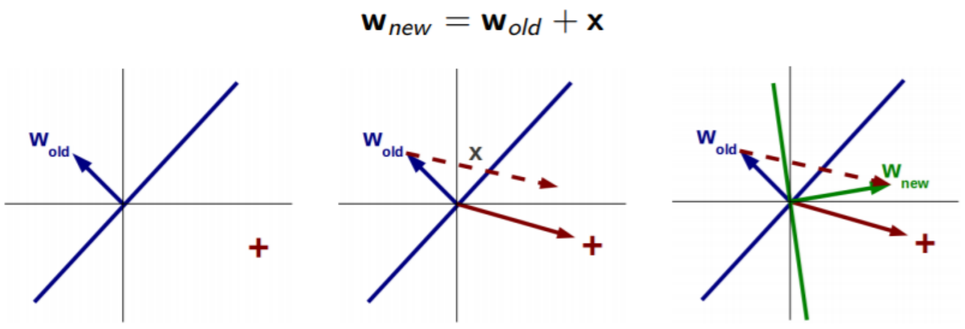


Fig. 8. Correction of actual positive misclassified instance.

  For $b = 0$, to classify a point x as positive, we simply want $\mathbf{w} \cdot \mathbf{x} > 0$, i.e. the angle between x and b to be acute. So in this case the weight vector should point towards positive instances.

- $y = 0$ but the perceptron thinks that $y_{pred} = 1 \Rightarrow \mathbf{w}_{old}\mathbf{x} + b_{old} > 0$.

  Then the weights and offset are corrected by

$$\mathbf{w}_{new} = \mathbf{w}_{old} - \mathbf{x}$$
$$b_{new} = b_{old} - 1$$

  Indeed, in this case for the new output

$$\begin{aligned} \mathbf{w}_{new} \cdot \mathbf{x} + b_{new} &= (\mathbf{w}_{old} - \mathbf{x})\mathbf{x} + b_{old} - 1 \\ &= \mathbf{w}_{old} \cdot \mathbf{x} + b_{old} - \mathbf{x} \cdot \mathbf{x} - 1 \\ &< \mathbf{w}_{old} \cdot \mathbf{x} + b_{old} \end{aligned}$$

The new output is more negative than the old, as desired. Below is how the correction works geometrically (again, for $b = 0$).

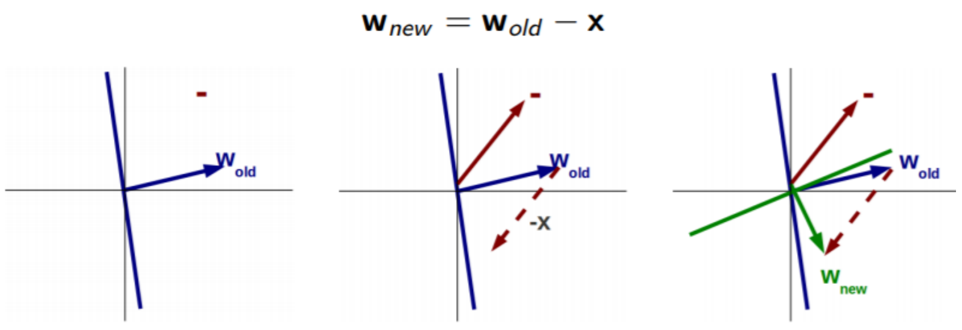


Fig. 9. Correction of actual negative misclassified instance.

Similarly, for $b = 0$, to classify a point $\mathbf{x}$ as negative we want $\mathbf{w} \cdot \mathbf{x} < 0$, i.e. the angle between $\mathbf{x}$ and $\mathbf{b}$ to be obtuse, i.e. the weight vector to point away from negative instances, as shown in the correction.

1.5 Perceptron weight correction - the algorithm

In reality, the weights and bias are not updated by $\mathbf{x}$ and 1 as this is too large and causes predicition errors to the other measurements, but by small fractions of them.

Definition 1.5. *The perceptron correction weight updates are*

$$\boldsymbol{x}_{new} = \boldsymbol{w}_{old} + \eta\, \boldsymbol{x} \tag{1.8}$$

$$b_{new} = b_{old} + \eta \tag{1.9}$$

η is called the l e a r n i n g r a t e.

Example 1.1. *We have the decision line $3x_1 + 4x_2 - 10 = 0$ and the positive point $\boldsymbol{x} = (1,1)$ that is misclassified. If the learning rate is $\eta = 0.1$, how many iterations will it take to correct it? (Credits: Udacity,* **TODO!** *)*

Solution 1.1. *Since the point to correct is $(4,5)$ and is actually negative $(y = 0)$, after one update we sum and the coefficients are*

$$w_1^{(1)} = 3 + 1\,\eta$$

$$w_2^{(1)} = 4 + 1\,\eta$$

$$b^{(1)} = -10 + \eta$$

And after n updates they are:

$$w_1^{(n)} = 3 + n\,\eta = 3 + 0.1n$$

$$w_2^{(n)} = 4 + n\,\eta = 4 + 0.1n$$

$$b^{(n)} = -10 + n\,\eta = -10 + 0.1n$$

We want to keep correcting until:

$$1w_1^{(n)} + 1w_2^{(n)} - 10 + 0.1n > 0 \Rightarrow \tag{1.10}$$

$$3 + 0.1n + 4 + 0.1n - 10 + 0.1n > 0 \Rightarrow \tag{1.11}$$

$$n > 10 \tag{1.12}$$

So at least 10 itearations are required to correct it.

That is the basis of the perceptron algorithm – iterate for each point, and if it is misclassified, update accordingly. It is described in pseudocode below. It can be proved that for linearly separable data the algorithm is guaranteed to converge.

Algorithm 1 The final perceptron algorithm.

```
1: procedure PERCEPTRON($\mathbf{x}_{1\times m}$, $\mathbf{y}_{1\times m}$, $\eta$, $N_{epochs}$)
2:     ▷ $\mathbf{x}$: a list of $m$ vectors (input data), $\mathbf{y}$: label (0 or 1) for each input. Each input also has $n$ features.
3:     ▷ $\eta$: learning rate, $N_{epochs}$: how many epochs (iterations) we want the algorithm to run for.
4:     $w_1$, $w_2$, ..., $w_n$, $b \leftarrow$ random values
5:     for $i = 0,\ldots,N_{epochs}$ do
6:         for $j = 1,\ldots,m$ do
7:             if $y_j = 0$ and $\mathbf{w}\cdot\mathbf{x}_j + b > 0$ then        ▷ misclassified negative point
8:                 $\mathbf{w} \leftarrow \mathbf{w} - \eta\,\mathbf{x}_j$
9:                 $b \leftarrow b - \eta$
10:            else if $y_j = 1$ and $\mathbf{w}\cdot\mathbf{x}_j + b < 0$ then   ▷ misclassified positive point
11:                $\mathbf{w} \leftarrow \mathbf{w} + \eta\,\mathbf{x}_j$
12:                $b \leftarrow b + \eta$
13:    return $\mathbf{w}$, $b$
```

1.6 Algorithm implementation

An implementation that learns the weights online (updates them as it reads the data) and runs for a certain number of epochs is listed in A.2. It is based on Udacity's skeleton code. In Fig. 10, where its result is visualised, the final boundary line is solid and the updates are dashed.

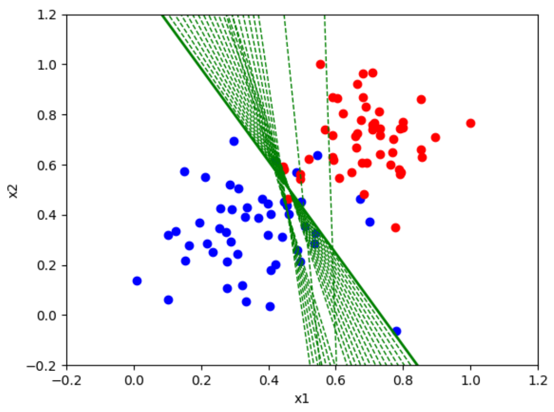


Fig. 10. Online training results for percetron online training. Positive data are blue and negative red.

A Appendices

A.1 Proof of Cor. 1.2 (perceptron boundary line)

The decision boundary has equation

$$w_1 x_1 + w_2 x_2 + b = 0 \tag{1}$$

$\mathbf{x}$ is a vector from the origin to some point (x_1, x_2) on the line. We are looking to minimise $||\mathbf{x}||$ under condition (1), where $\mathbf{x} = (x_1, x_2)$. Eq. (1) can be re-written as:

$$\mathbf{w} \cdot \mathbf{x} = -b \tag{2}$$

, where $\mathbf{w} \cdot \mathbf{x} = ||\mathbf{w}||\,||\mathbf{x}||\,cos(\theta)$ is the dot product, θ the angle between the two vectors, and $w = (w_1, w_2)$ the weight vector perpendicular to the line. Therefore, and since $||x|| \geq 0$, Eq. (2) can be solved for $||x||$ as:

$$||\mathbf{x}|| = \frac{|b|}{||\mathbf{w}||\,|\cos(\theta)|} \Rightarrow$$

$$||\mathbf{x}|| \leq \frac{|b|}{||\mathbf{w}||}$$

The equality is achieved when $|\cos(\theta)| = 1$, i.e. when $\mathbf{x}$ and $\mathbf{w}$ are parallel or anti-parallel. The minimum distance of $\mathbf{x}$ from the origin is therefore $\frac{|b|}{||\mathbf{w}||}$. □

A.2 Perceptron with online training in Python.

```python
import numpy as np
import csv
import matplotlib.pyplot as plt
# Setting the random seed, feel free to change it and see different solutions.
np.random.seed(42)


def stepFunction(t):
    if t >= 0:
        return 1
    return 0


def prediction(X, W, b):
    return stepFunction((np.matmul(X,W)+b)[0])


# TODO: Fill in the code below to implement the perceptron trick.
# The function should receive as inputs the data X, the labels y,
# the weights W (as an array), and the bias b,
# update the weights and bias W, b, according to the perceptron algorithm,
# and return W and b.
def perceptronStep(X, y, W, b, learn_rate = 0.01):
    for i, XX in enumerate(X):
        y_pred = prediction(X[i], W, b)
        #print(y_pred, y[i])
        if y[i] == 1 and y_pred == 0:
            W[0] += X[i][0] * learn_rate
            W[1] += X[i][1] * learn_rate
            b += 1 * learn_rate
        elif y[i] == 0 and y_pred == 1:
            W[0] -= X[i][0] * learn_rate
            W[1] -= X[i][1] * learn_rate
            b -= 1 * learn_rate
    return W, b


# This function runs the perceptron algorithm repeatedly on the dataset,
# and returns a few of the boundary lines obtained in the iterations,
# for plotting purposes.
# Feel free to play with the learning rate and the num_epochs,
# and see your results plotted below.
def trainPerceptronAlgorithm(X, y, learn_rate = 0.01, num_epochs = 25):
    x_min, x_max = min(X.T[0]), max(X.T[0])
    y_min, y_max = min(X.T[1]), max(X.T[1])
    W = np.array(np.random.rand(2,1))
    b = np.random.rand(1)[0] + x_max
    # These are the solution lines that get plotted below.
    boundary_lines = []
    for i in range(num_epochs):
        # In each epoch, we apply the perceptron step.
        W, b = perceptronStep(X, y, W, b, learn_rate)
        boundary_lines.append((-W[0]/W[1], -b/W[1]))
    return boundary_lines


def readData(fpath = 'data.csv'):
    """
        Reads input data, each line is of the form:
        x1, x2, y
        Note that y = 0 or 1
    """
    with open(fpath) as csvf:
        X = []
        y = []
        csv_data= csv.reader(csvf, delimiter = ',')
        for row in  csv_data:
            X.append([float(row[0]), float(row[1])])
            y.append(float(row[2]))
    X = np.array(X, np.float32)
    y = np.array(y, np.float32)
    return X, y


def plotData(X, y):
    """
        Plots what is returned by readData
    """
    for i in  range(len(X)):
        if y[i] == 1:
            plt.plot(X[i][0], X[i][1], "o", color = 'blue')
        elif y[i] == 0:
            plt.plot(X[i][0], X[i][1], "o", color = 'red')


def plotBoundaries(bound_coeffs):
    """
        Plots the straight lines returned by trainPerceptronAlgorithm
    """
    plt.xlim((-0.2,1.2))
    plt.ylim((-0.2,1.2))
    # boundary_lines.append((-W[0]/W[1], -b/W[1]))
    for i in  range(len(bound_coeffs)):
        x = np.linspace(0, 1, 50)
        y = bound_coeffs[i][0]*x +  bound_coeffs[i][1]
        if i != len(bound_coeffs) - 1:
            plt.plot(x, y, 'g--', lw=1)
        else:
            plt.plot(x, y, 'g-', lw=2)


def main():
    X, y = readData()
    plotData(X,y)
    boundaries = trainPerceptronAlgorithm(X,y)
    plotBoundaries(boundaries)
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.show()


main()
```