**Lab Assignment – 3**

**Machine Learning Lab**

**Name : Bandla Lavanya**

**Regd Num : 19MCB0008**

**Topic:Neural networks**

**Perceptron:**

Writing a very simple Neural Network implementing the logical "And" and "Or" functions.

Starting with the "And" function. It is defined for two inputs:

Input1 Input2 Output

0 0 0

0 1 0

1 0 0

1 1 1

**Using AND and OR gate function for line separation:**

Considering that we have two attributes describing an eddible object like a fruit.

for example: "sweetness" and "sourness"

Describing this by points in a two-dimensional space. The x axis for the sweetness and the y axis for the sourness. Imagine now that we have two fruits as points in this space, i.e. an orange at position (3.5, 1.8) and a lemon at (1.1, 3.9).

Dividing lines to define the points which are more lemon-like and which are more orange-like. The following program calculates and renders a bunch of lines. The red ones are completely unusable for this purpose, because they are not separating the classes. Yet, it is obvious that even the green ones are not all useful.

Using AND and OR gate for line separation

**Source code:**

import numpy as np

import matplotlib.pyplot as plt # AND GATE for line seperation

def create\_distance\_function(a, b, c):

""" 0 = ax + by + c """

def distance(x, y):

""" returns tuple (d, pos)

d is the distance

If pos == -1 point is below the line,

0 on the line and +1 if above the line

"""

nom = a \* x + b \* y + c

if nom == 0:

pos = 0

elif (nom<0 and b<0) or (nom>0 and b>0):

pos = -1

else:

pos = 1

return (np.absolute(nom) / np.sqrt( a \*\* 2 + b \*\* 2), pos)

return distance

points = [ (3.5, 1.8), (1.1, 3.9) ]

fig, ax = plt.subplots()

ax.set\_xlabel("sweetness")

ax.set\_ylabel("sourness")

ax.set\_xlim([-1, 6])

ax.set\_ylim([-1, 8])

X = np.arange(-0.5, 5, 0.1)

colors = ["r", ""] # for the samples

size = 10

for (index, (x, y)) in enumerate(points):

if index== 0:

ax.plot(x, y, "o",

color="darkorange",

markersize=size)

else:

ax.plot(x, y, "oy",

markersize=size)

step = 0.05

for x in np.arange(0, 1+step, step):

slope = np.tan(np.arccos(x))

dist4line1 = create\_distance\_function(slope, -1, 0)

#print("x: ", x, "slope: ", slope)

Y = slope \* X

results = []

for point in points:

results.append(dist4line1(\*point))

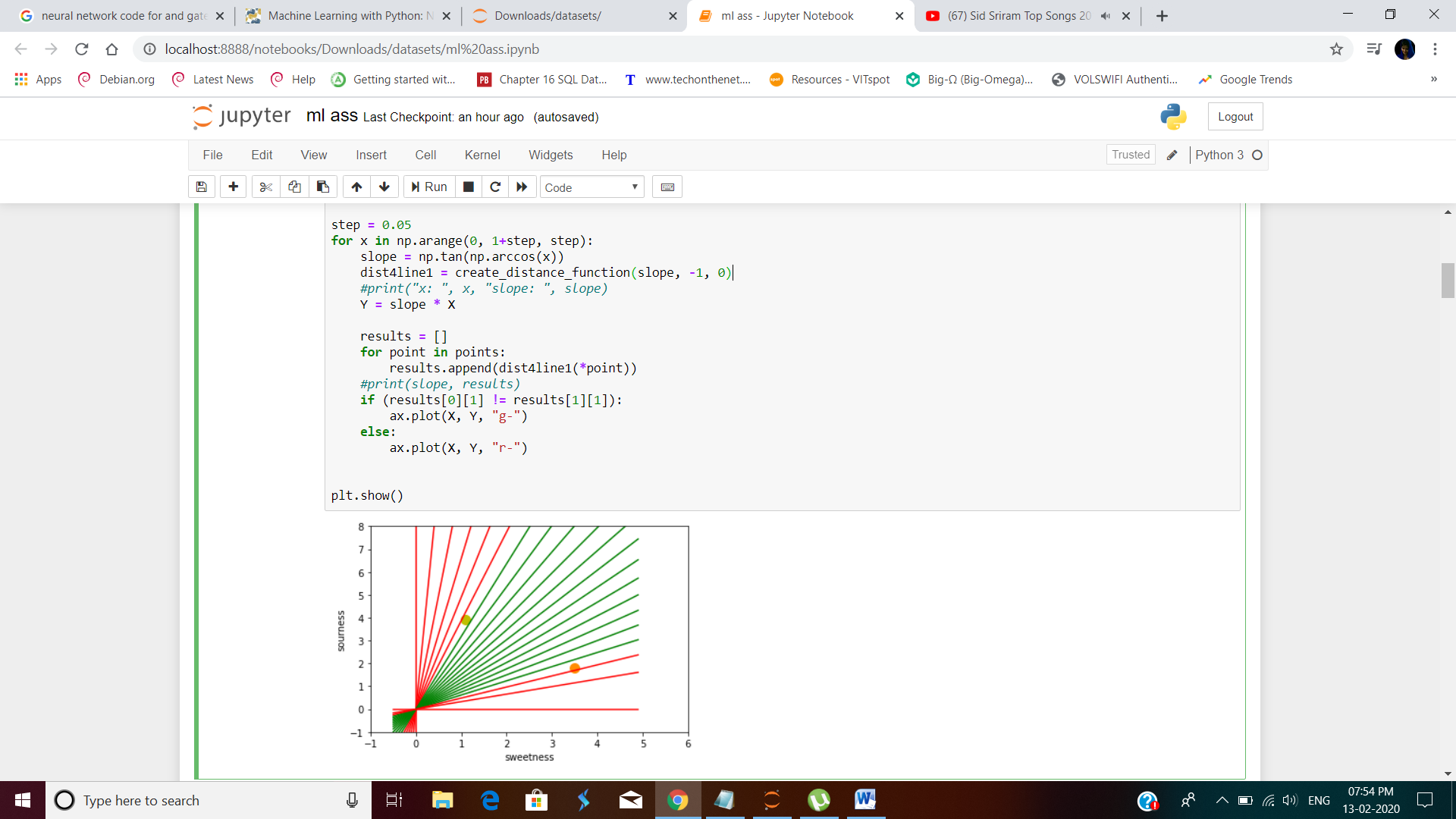
if (results[0][1] != results[1][1]):

ax.plot(X, Y, "g-")

else:

ax.plot(X, Y, "r-")

plt.show()



In the following program, we train a neural network to classify two clusters in a 2-dimensional space. It is shown that we have two classes class1 and class2. We will create those points randomly with the help of a line, the points of class2 will be above the line and the points of class1 will be below the line.

**Source code:**

import numpy as np

**class Perceptron:**

def \_\_init\_\_(self, input\_length, weights=None):

if weights is None:

self.weights = np.ones(input\_length) \* 0.5

else:

self.weights = weights

@staticmethod

def unit\_step\_function(x):

if x > 0.5:

return 1

return 0

def \_\_call\_\_(self, in\_data):

weighted\_input = self.weights \* in\_data

weighted\_sum = weighted\_input.sum()

return Perceptron.unit\_step\_function(weighted\_sum)

p = Perceptron(2, np.array([0.5, 0.5]))

data\_in = np.empty((2,))

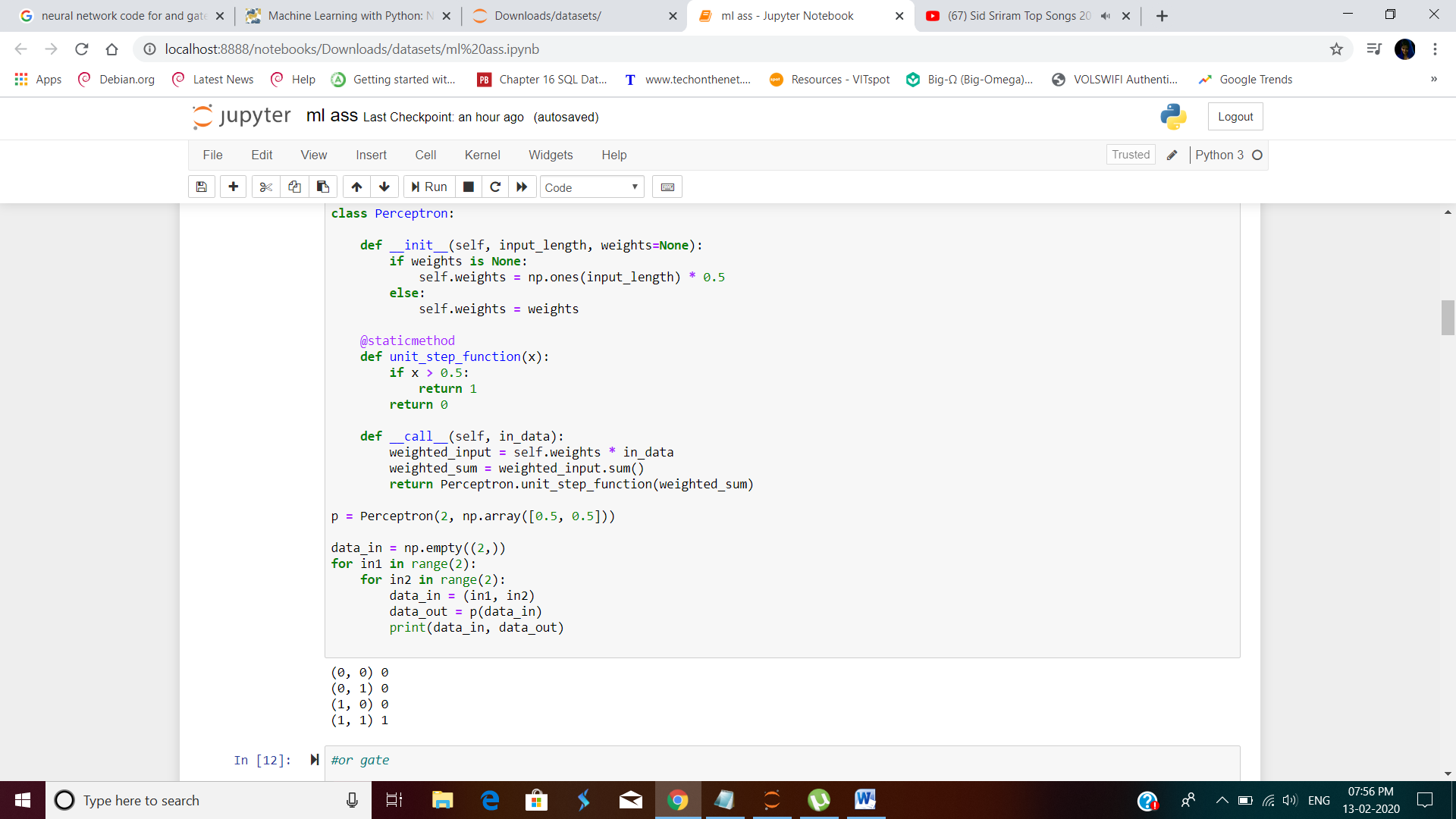
for in1 in range(2):

for in2 in range(2):

data\_in = (in1, in2)

data\_out = p(data\_in)

print(data\_in, data\_out)



We will see that the neural network will find a line that separates the two classes. This line should not be mistaken for the line, which we used to create the points.

This line is called a **decision boundary**.

**OR gate**

**Source code:**

import numpy as np

from collections import Counter

class Perceptron:

def \_\_init\_\_(self, input\_length, weights=None):

if weights==None:

self.weights = np.random.random((input\_length)) \* 2 - 1

self.learning\_rate = 0.1

@staticmethod

def unit\_step\_function(x):

if x < 0:

return 0

return 1

def \_\_call\_\_(self, in\_data):

weighted\_input = self.weights \* in\_data

weighted\_sum = weighted\_input.sum()

return Perceptron.unit\_step\_function(weighted\_sum)

def adjust(self,

target\_result,

calculated\_result,

in\_data):

error = target\_result - calculated\_result

for i in range(len(in\_data)):

correction = error \* in\_data[i] \*self.learning\_rate

self.weights[i] += correction

def above\_line(point, line\_func):

x, y = point

if y > line\_func(x):

return 1

else:

return 0

points = np.random.randint(1, 100, (100, 2))

p = Perceptron(2)

def lin1(x):

return x + 4

for point in points:

p.adjust(above\_line(point, lin1),

p(point),

point)

evaluation = Counter()

for point in points:

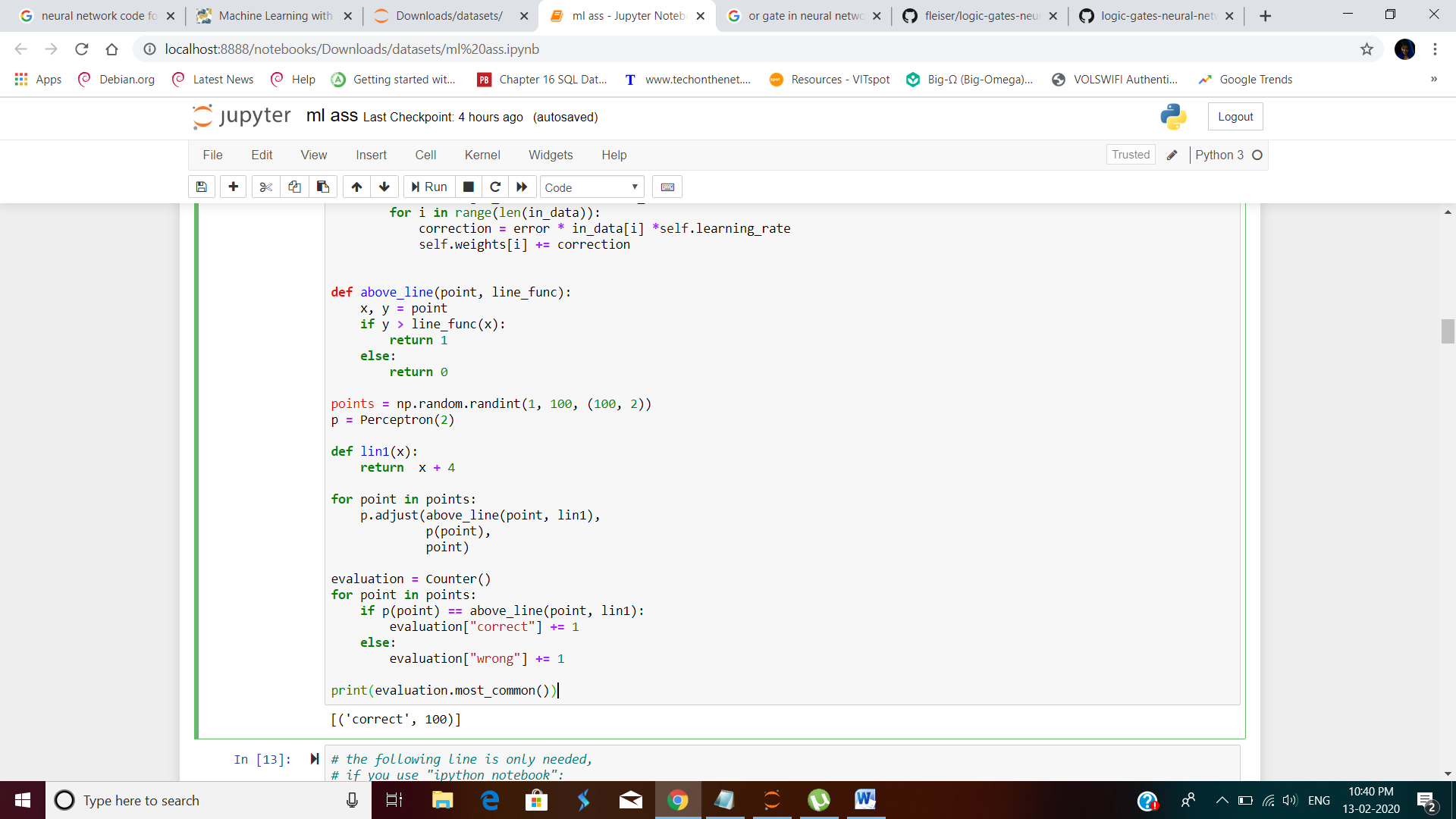
if p(point) == above\_line(point, lin1):

evaluation["correct"] += 1

else:

evaluation["wrong"] += 1

print(evaluation.most\_common())



For identifying the group of clusters by separating from the line

**Source code:**

%matplotlib inline

from matplotlib import pyplot as plt

cls = [[], []]

for point in points:

cls[above\_line(point, lin1)].append(tuple(point))

colours = ("r", "b")

for i in range(2):

X, Y = zip(\*cls[i])

plt.scatter(X, Y, c=colours[i])

X = np.arange(-3, 120)

m = -p.weights[0] / p.weights[1]

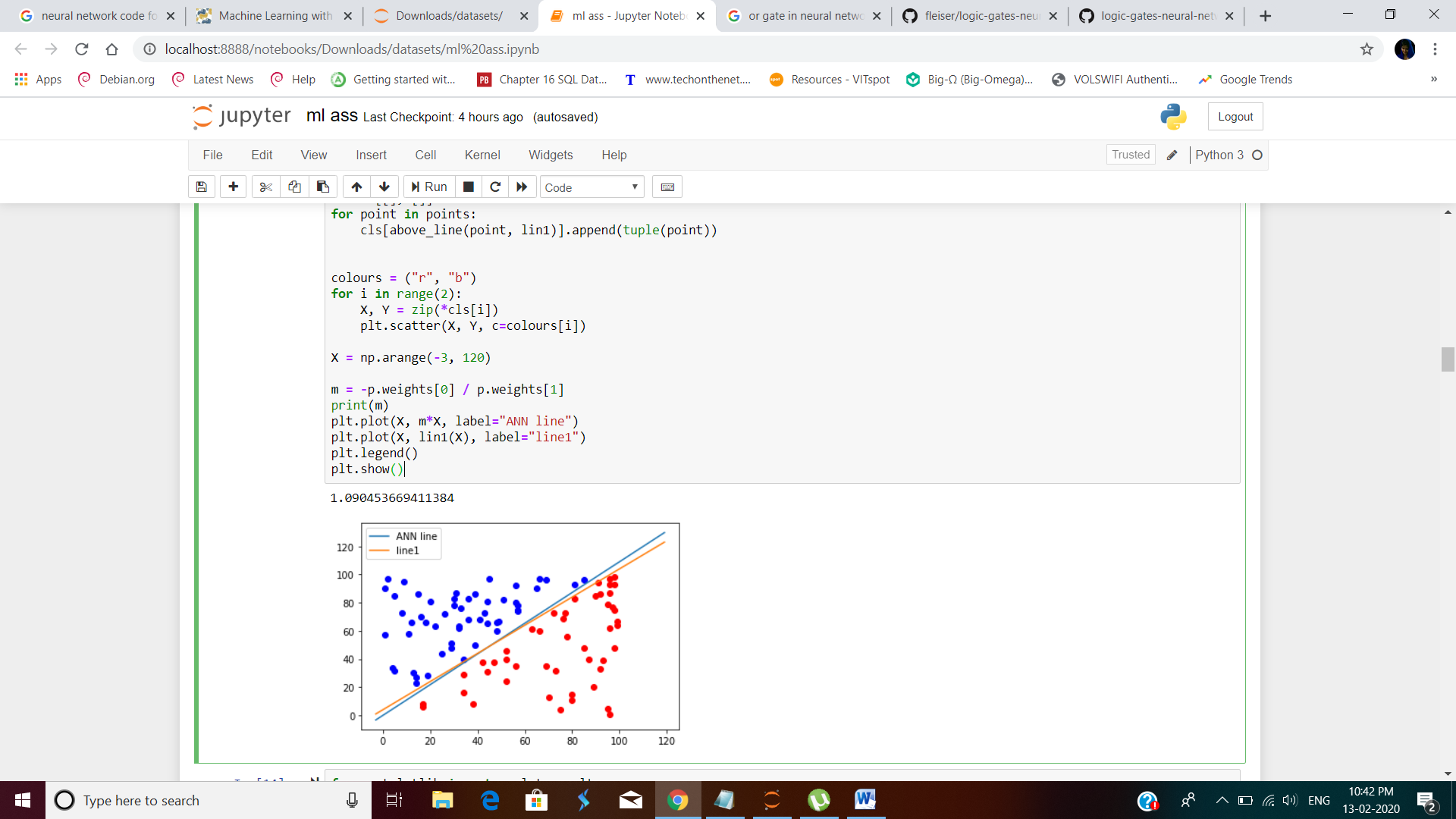
print(m)

plt.plot(X, m\*X, label="ANN line")

plt.plot(X, lin1(X), label="line1")

plt.legend()

plt.show()



As the constant term bb determines the point at which a line crosses the y-axis, i.e. the y-intercept, we can see that our network can only calculate lines which pass through the origin, i.e. the point (0, 0). We will need a bias to get other lines as well, i.e. lines which don't go through the origin. A neural network with bias nodes can look like this:

Now, the linear equation for a perceptron contains a bias:

Creating a new dataset for our next experiments

**Source code:**

from matplotlib import pyplot as plt

class1 = [(3, 4), (4.2, 5.3), (4, 3), (6, 5), (4, 6), (3.7, 5.8),

(3.2, 4.6), (5.2, 5.9), (5, 4), (7, 4), (3, 7), (4.3, 4.3) ]

class2 = [(-3, -4), (-2, -3.5), (-1, -6), (-3, -4.3), (-4, -5.6),

(-3.2, -4.8), (-2.3, -4.3), (-2.7, -2.6), (-1.5, -3.6),

(-3.6, -5.6), (-4.5, -4.6), (-3.7, -5.8) ]

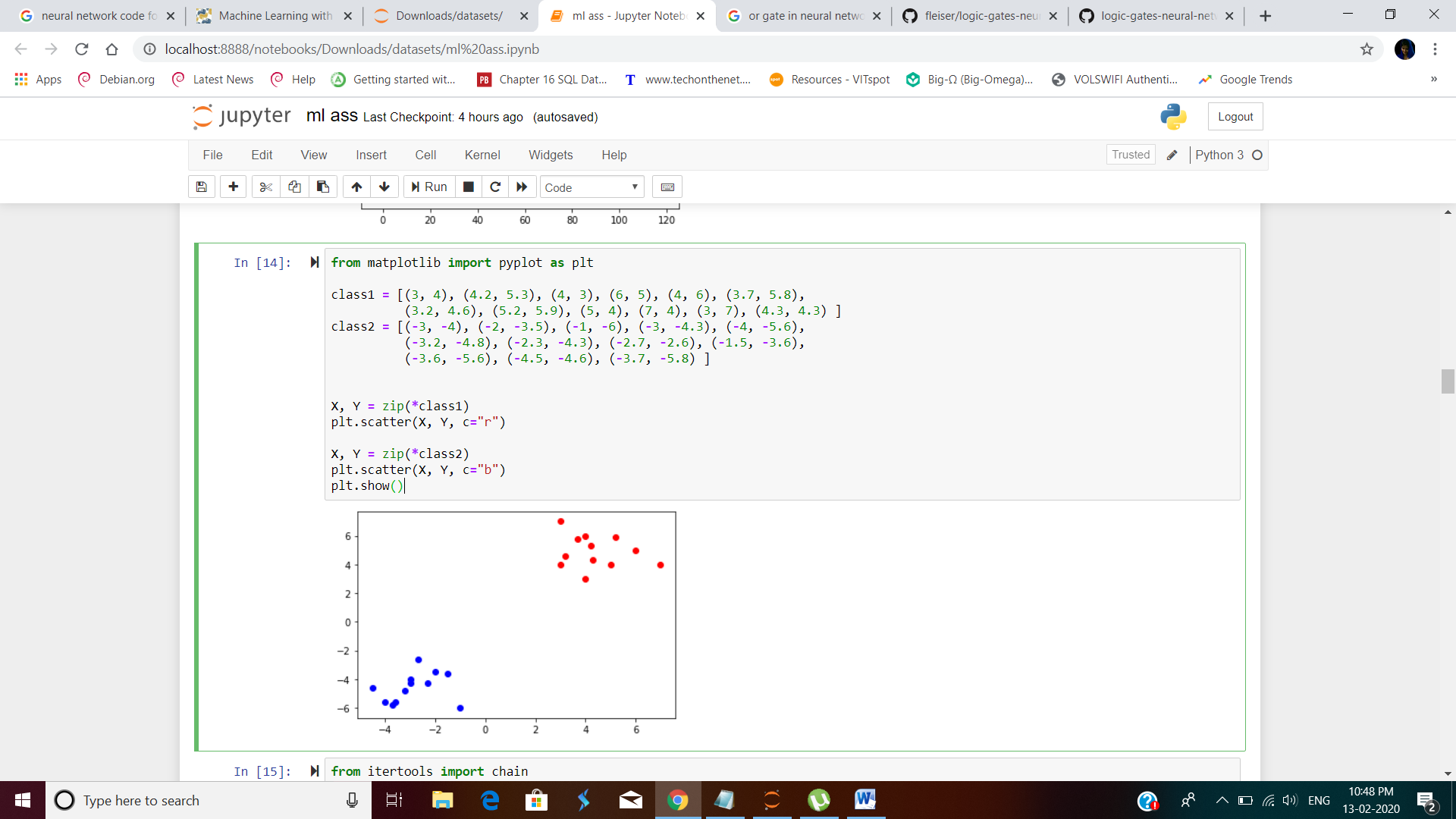
X, Y = zip(\*class1)

plt.scatter(X, Y, c="r")

X, Y = zip(\*class2)

plt.scatter(X, Y, c="b")

plt.show()



**The XOR (exclusive or) function is defined by the following truth table:**

**Neural networks with bias value:**

Considering our initial example with the random points above and below a line. We will rewrite the code using a bias value.

First we will create two classes with random points, which are not separable by a line crossing the origin.

We will add a bias b to our neural network. This leads us to the following condition

x1w1+x2w2+bw3=0x1w1+x2w2+bw3=0

We can change the equation into

x2=−w1w2x1−w3w2b

**source code:**

import numpy as np

from collections import Counter

class Perceptron:

def \_\_init\_\_(self, input\_length, weights=None):

if weights==None:

self.weights = np.random.random((input\_length)) \* 2 - 1

self.learning\_rate = 0.1

@staticmethod

def unit\_step\_function(x):

if x < 0:

return 0

return 1

def \_\_call\_\_(self, in\_data):

weighted\_input = self.weights \* in\_data

weighted\_sum = weighted\_input.sum()

return Perceptron.unit\_step\_function(weighted\_sum)

def adjust(self,

target\_result,

calculated\_result,

in\_data):

error = target\_result - calculated\_result

for i in range(len(in\_data)):

correction = error \* in\_data[i] \*self.learning\_rate

self.weights[i] += correction

p = Perceptron(2)

for point, label in learnset:

p.adjust(label,

p(point),

point)

evaluation = Counter()

for point, label in learnset:

if p(point) == label:

evaluation["correct"] += 1

else:

evaluation["wrong"] += 1

print(evaluation.most\_common())

colours = ["b", "r"]

for i in range(2):

plt.scatter(X[i], Y[i], c=colours[i])

XR = np.arange(-8, 4)

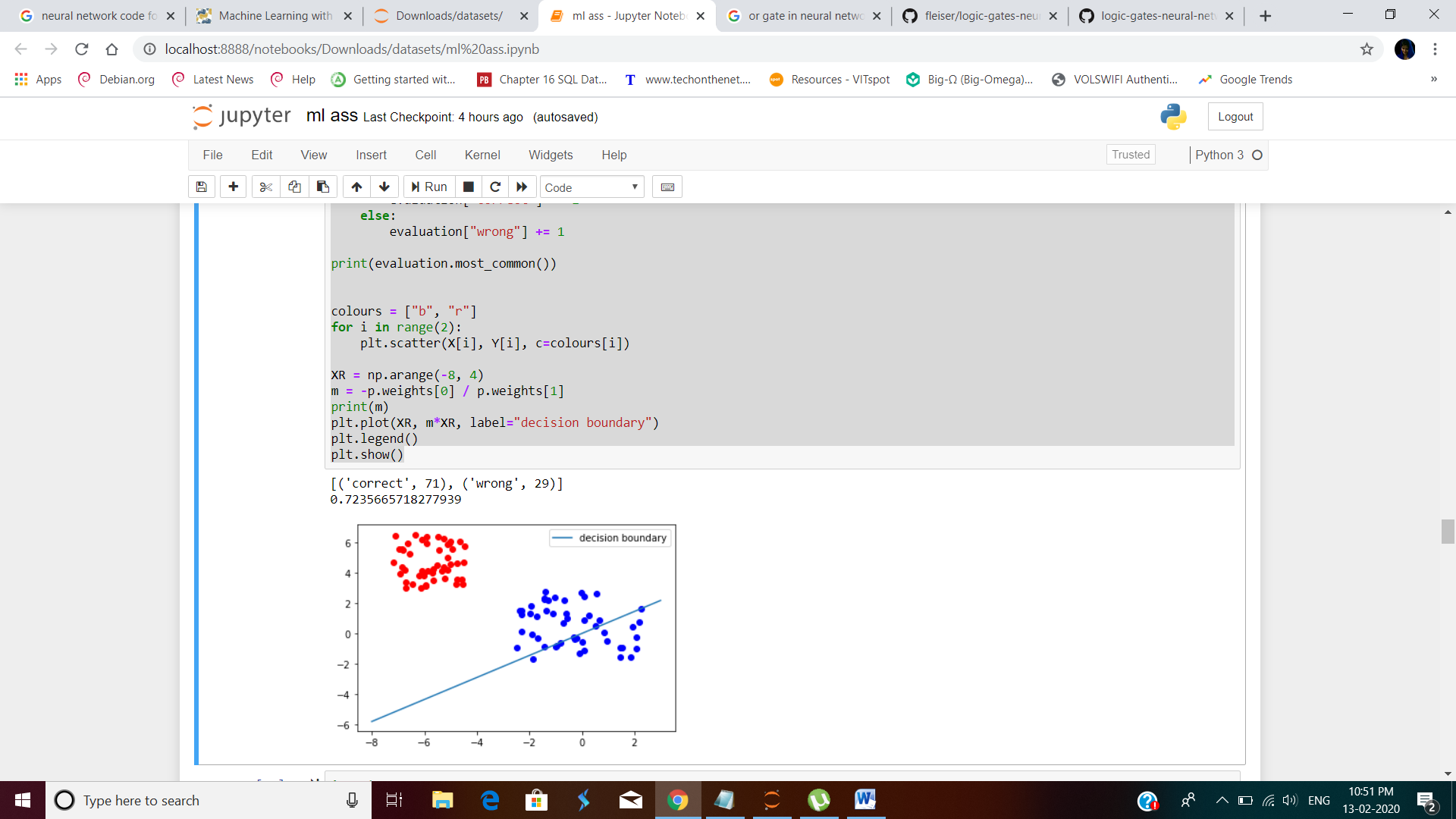
m = -p.weights[0] / p.weights[1]

print(m)

plt.plot(XR, m\*XR, label="decision boundary")

plt.legend()

plt.show()



**It is not possible to find a solution with one neuron and without a bias node**. The reason is that the class of the blue data points spread around the origin. Without bias nodes we get only lines going through the origin as we have mentioned earlier. It is easy to see that no line going through the origin can separate the blue from the red data.

The following class uses bias nodes and solves this problem:

**Source code:**

import numpy as np

from collections import Counter

class Perceptron:

def \_\_init\_\_(self, input\_length, weights=None):

if weights==None:

# input\_length + 1 because bias needs a weight as well

self.weights = np.random.random((input\_length + 1)) \* 2 - 1

self.learning\_rate = 0.05

self.bias = 1

@staticmethod

def sigmoid\_function(x):

res = 1 / (1 + np.power(np.e, -x))

return 0 if res < 0.5 else 1

def \_\_call\_\_(self, in\_data):

weighted\_input = self.weights[:-1] \* in\_data

weighted\_sum = weighted\_input.sum() + self.bias \*self.weights[-1]

return Perceptron.sigmoid\_function(weighted\_sum)

def adjust(self,

target\_result,

calculated\_result,

in\_data):

error = target\_result - calculated\_result

for i in range(len(in\_data)):

correction = error \* in\_data[i] \*self.learning\_rate

#print("weights: ", self.weights)

#print(target\_result, calculated\_result, in\_data, error, correction)

self.weights[i] += correction

# correct the bias:

correction = error \* self.bias \* self.learning\_rate

self.weights[-1] += correction

p = Perceptron(2)

for point, label in learnset:

p.adjust(label,

p(point),

point)

evaluation = Counter()

for point, label in learnset:

if p(point) == label:

evaluation["correct"] += 1

else:

evaluation["wrong"] += 1

print(evaluation.most\_common())

colours = ["b", "r"]

for i in range(2):

plt.scatter(X[i], Y[i], c=colours[i])

XR = np.arange(-8, 4)

m = -p.weights[0] / p.weights[1]

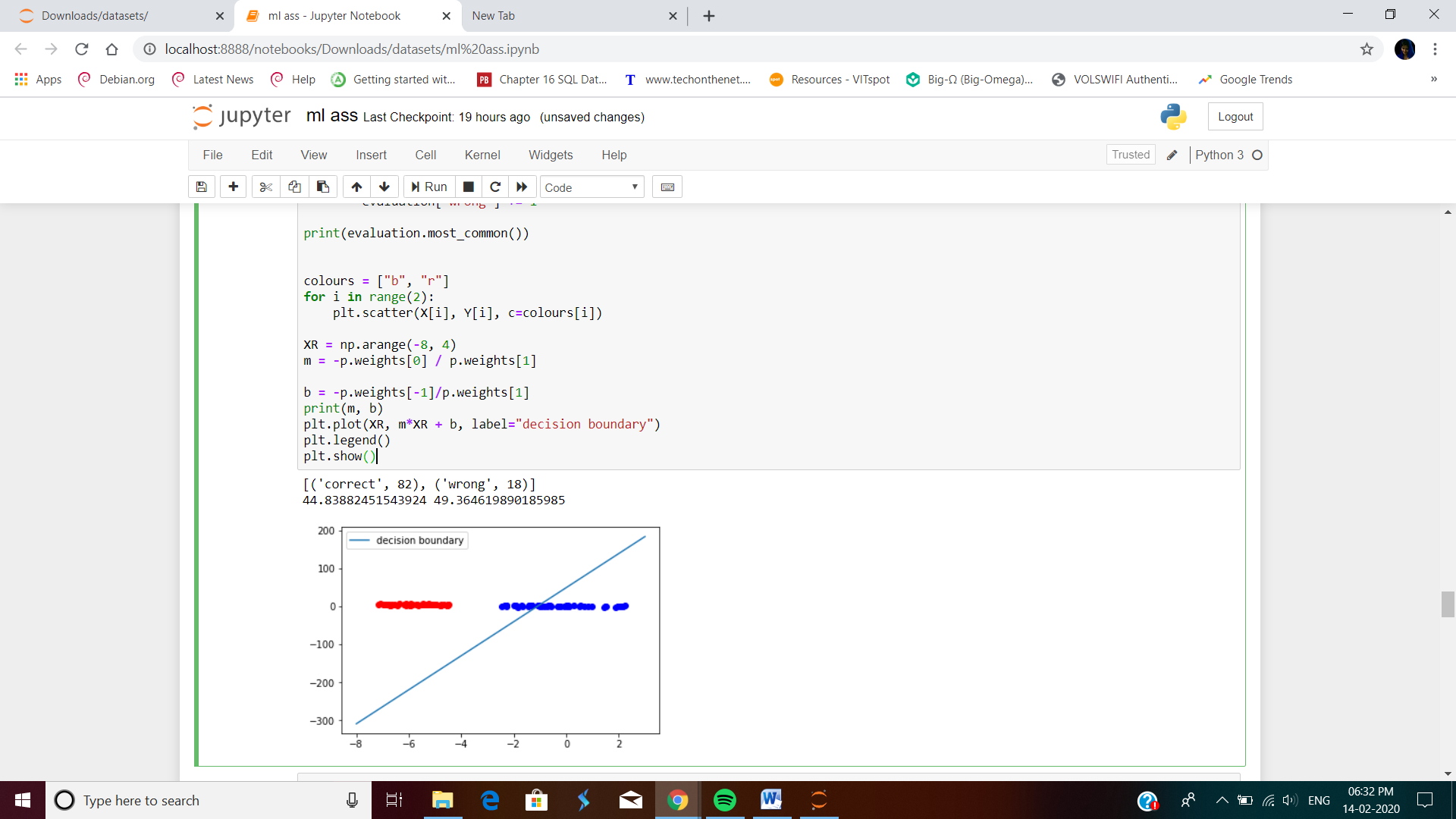
b = -p.weights[-1]/p.weights[1]

print(m, b)

plt.plot(XR, m\*XR + b, label="decision boundary")

plt.legend()

plt.show()



Logic gates namely AND, OR, NOT or NAND are some of the building blocks of every technological breakthrough for the past decade specially for hardware. I am going to represent Logic Gates using the basics of Neural Network. I’ve created a perceptron using numpy that implements this Logic Gates with the dataset acting as the input to the perceptron.

**Source code for the AND ,OR , NAND gate functions.**

import numpy

import scipy.special

import glob

import scipy.misc

class neuralNetwork:

def \_\_init\_\_(self, inputNodes, hiddenOneNodes, hiddenTwoNodes, hiddenThreeNodes, finalNodes, alpha):

self.inputNodes = inputNodes

self.hiddenOneNodes = hiddenOneNodes

self.hiddenTwoNodes = hiddenTwoNodes

self.hiddenThreeNodes = hiddenThreeNodes

self.finalNodes = finalNodes

self.alpha = alpha

self.weightsInputHidden = numpy.random.normal(0.0, pow(self.hiddenOneNodes, -0.5),(self.hiddenOneNodes,self.inputNodes))

self.weightsHiddenOneHiddenTwo = numpy.random.normal(0.0, pow(self.hiddenTwoNodes,-0.5),(self.hiddenTwoNodes,self.hiddenOneNodes))

self.weightsHiddenTwoHiddenThree = numpy.random.normal(0.0, pow(self.hiddenThreeNodes,-0.5),(self.hiddenThreeNodes,self.hiddenTwoNodes))

self.weightsHiddenOutput = numpy.random.normal(0.0, pow(self.hiddenOneNodes,-0.5),(self.finalNodes, self.hiddenThreeNodes))

pass

def train(self, inputs, target):

inputs = numpy.array(inputs, ndmin=2).T

target = numpy.array(target, ndmin=2).T

hiddenInput = numpy.dot(self.weightsInputHidden,inputs)

hiddenOneOutput = self.sigmoid(hiddenInput)

hiddenTwoInput = numpy.dot(self.weightsHiddenOneHiddenTwo,hiddenOneOutput)

hiddenTwoOutput = self.sigmoid(hiddenTwoInput)

hiddenThreeInput = numpy.dot(self.weightsHiddenTwoHiddenThree,hiddenTwoOutput)

hiddenThreeOutput = self.sigmoid(hiddenThreeInput)

finalInput = numpy.dot(self.weightsHiddenOutput,hiddenThreeOutput)

finalOutput = self.sigmoid(finalInput)

outputError = target - finalOutput

hiddenOutputError = numpy.dot(self.weightsHiddenOutput.T, outputError)

hiddenThreeHiddenTwoError = numpy.dot(self.weightsHiddenTwoHiddenThree.T, hiddenOutputError)

hiddenTwoHiddenOneError = numpy.dot(self.weightsHiddenOneHiddenTwo.T, hiddenThreeHiddenTwoError)

hiddenInputError = numpy.dot(self.weightsInputHidden.T, hiddenTwoHiddenOneError)

self.weightsHiddenOutput += self.alpha \* numpy.dot((outputError \* finalOutput \* (1.0 - finalOutput)),numpy.transpose(hiddenThreeOutput))

self.weightsHiddenTwoHiddenThree += self.alpha \* numpy.dot((hiddenOutputError \* hiddenThreeOutput \* (1.0 - hiddenThreeOutput)),numpy.transpose(hiddenTwoOutput))

self.weightsHiddenOneHiddenTwo += self.alpha \* numpy.dot((hiddenThreeHiddenTwoError \* hiddenTwoOutput \* (1.0 - hiddenTwoOutput)),numpy.transpose(hiddenOneOutput))

self.weightsInputHidden += self.alpha \* numpy.dot((hiddenTwoHiddenOneError \* hiddenOneOutput \* (1.0 - hiddenOneOutput)),numpy.transpose(inputs))

pass

def query(self, inputs):

inputs = numpy.array(inputs, ndmin=2).T

hiddenInput = numpy.dot(self.weightsInputHidden,inputs)

hiddenOneOutput = self.sigmoid(hiddenInput)

hiddenTwoInput = numpy.dot(self.weightsHiddenOneHiddenTwo,hiddenOneOutput)

hiddenTwoOutput = self.sigmoid(hiddenTwoInput)

hiddenThreeInput = numpy.dot(self.weightsHiddenTwoHiddenThree,hiddenTwoOutput)

hiddenThreeOutput = self.sigmoid(hiddenThreeInput)

finalInput = numpy.dot(self.weightsHiddenOutput,hiddenThreeOutput)

finalOutput = self.sigmoid(finalInput)

return finalOutput

pass

def sigmoid(self, x):

return scipy.special.expit(x)

pass

AND GATE:

Given training data to the neural network and predicting the output after and before training the data.

**Source code:**

n = neuralNetwork(2,12,36,12,1,0.1)

print('Before training')

print(n.query([0,0]))

print(n.query([0,1]))

print(n.query([1,0]))

print(n.query([1,1]))

print("Training...")

for i in range(0, 10000):

n.train([0,0],[0])

n.train([0,1],[0])

n.train([1,0],[0])

n.train([1,1],[1])

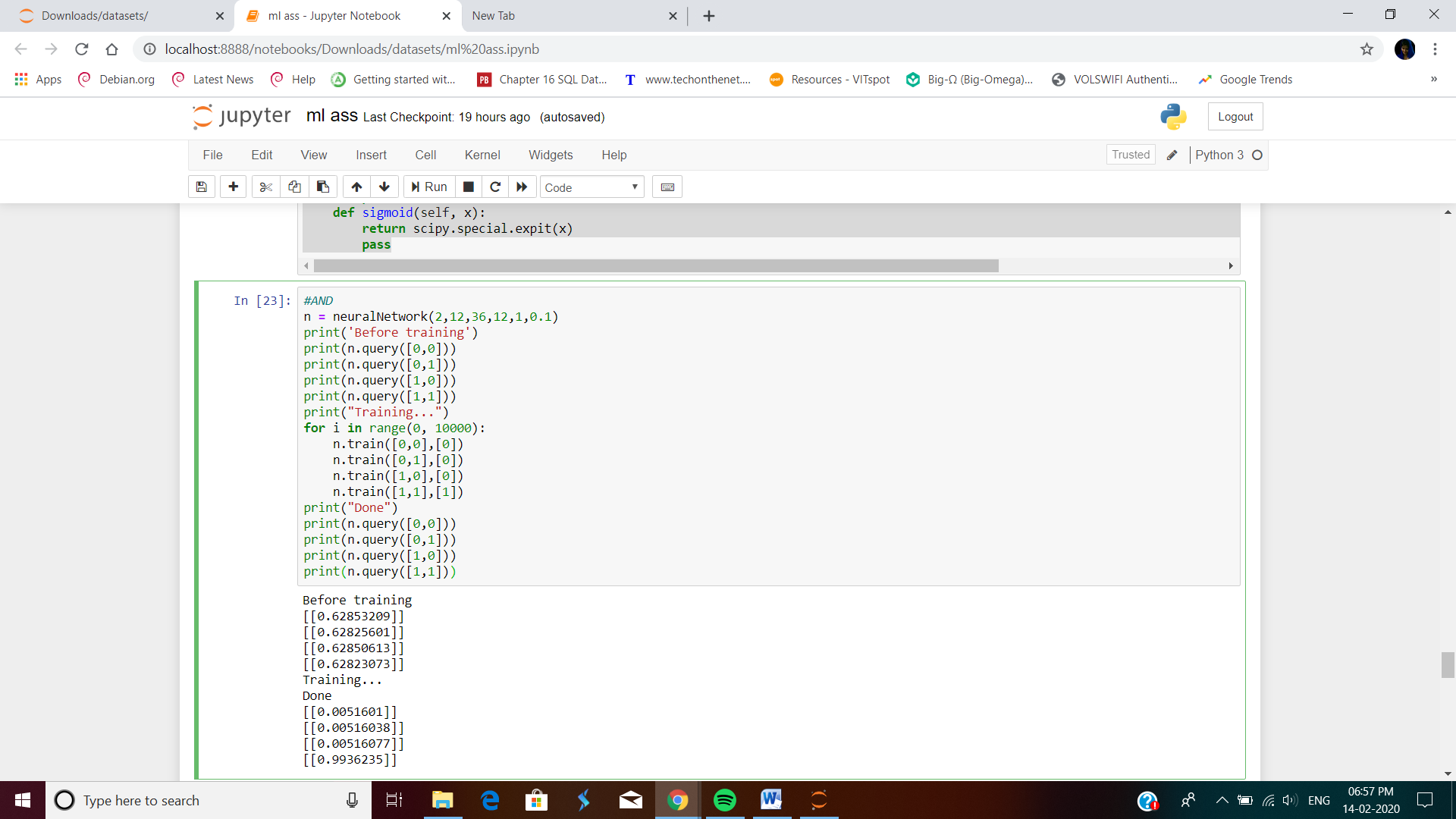
print("Done")

print(n.query([0,0]))

print(n.query([0,1]))

print(n.query([1,0]))

print(n.query([1,1]))



**Or gate function:**

**Source code:**

n = neuralNetwork(2,12,36,12,1,0.1)

print('Before training')

print(n.query([0,0]))

print(n.query([0,1]))

print(n.query([1,0]))

print(n.query([1,1]))

print("Training...")

for i in range(0, 10000):

n.train([0,0],[0])

n.train([0,1],[1])

n.train([1,0],[1])

n.train([1,1],[1])

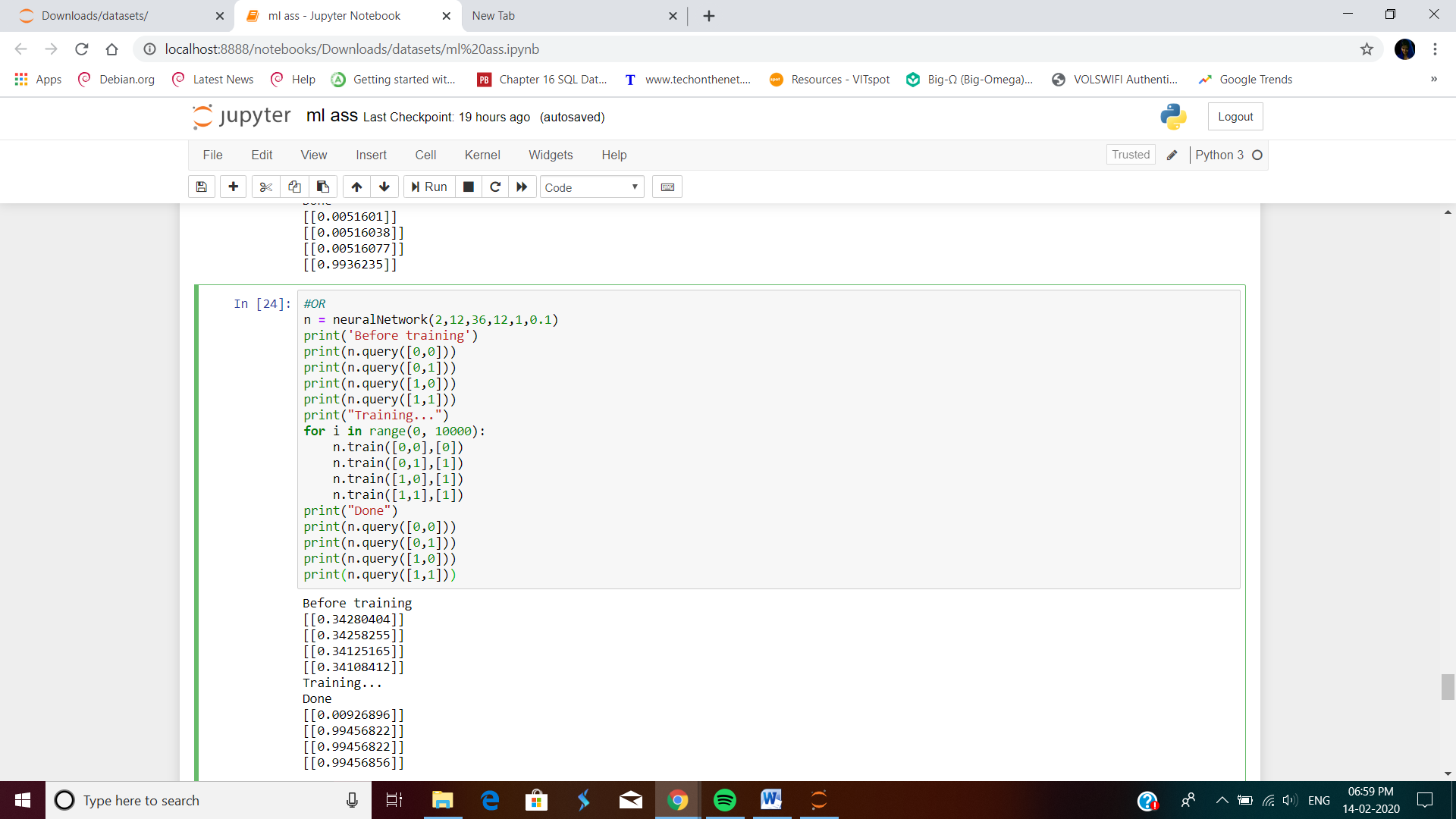
print("Done")

print(n.query([0,0]))

print(n.query([0,1]))

print(n.query([1,0]))

print(n.query([1,1]))



**NAND gate function:**

**Source code:**n = neuralNetwork(2,12,36,12,1,0.1)

print('Before training')

print(n.query([0,0]))

print(n.query([0,1]))

print(n.query([1,0]))

print(n.query([1,1]))

print("Training...")

for i in range(0, 10000):

n.train([0,0],[1])

n.train([0,1],[1])

n.train([1,0],[1])

n.train([1,1],[0])

print("Done")

print(n.query([0,0]))

print(n.query([0,1]))

print(n.query([1,0]))

print(n.query([1,1]))

